Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Luka Šarc

Prožna oblačna arhitektura aplikacij z uporabo prekinjevalcev toka

magistrsko delo

magistrski program druge stopnje
računalništvo in informatika

Mentor: prof. dr. Matjaž Branko Jurič

Ljubljana, 2017

Zahvala

Zahvaljujem se mentorju prof. dr. Matjažu Branku Juriču za usmerjanje in svetovanje pri izdelavi magistrskega dela. Hvala kolegom in sodelavcem iz laboratorija LIIS za strokovno pomoč, številne vesele in zabavne trenutke ter dobro vzdušje v ekipi. Hvala družini za podporo, razumevanje in pomoč tekom celotnega študija. Hvala prijateljem ter kolegom, s katerimi smo skupaj prebredli skozi študij. Hvala, Ana, za vso moralno podporo, razumevanje in številne lepe trenutke, ki so lajšali težje.

Luka Šarc, 2017

Kazalo

Povzetek

Abstract

1 Uvod **1**

1.1 Motivacija 1

1.2 Cilji 3

1.3 Struktura 4

2 Arhitektura oblačnih aplikacij **5**

2.1 Kratek pregled arhitekture oblačnih aplikacij 6

2.2 Arhitektura mikrostoritev 9

2.3 Ključni koncepti arhitekture mikrostoritev 11

2.4 Pregled ogrodij za razvoj mikrostoritev v programskem jeziku Java 15

2.5 Vzorci komunikacije med mikrostoritvami in potreba po odpornosti na napake 21

3 Odpornost na napake **25**

3.1 Stabilnost ob napakah 25

3.2 Orodja za odpornost na napake v programskem jeziku Java . . 32

4 Rešitev za doseganje odpornosti na napake v Javi **41**

4.1 Zasnova in implementacija rešitve 41

4.2 Umestitev orodja Hystrix 44

4.3 Konfiguracija odpornosti na napake 47
4.4 Metrike odpornosti na napake 49
4.5 Primer uporabe rešitve KumuluzEE Fault Tolerance 52

5 Skalabilnost in izvajanje v oblačnih aplikacijah 57
5.1 Skalabilnost v oblačnih aplikacijah 57
5.2 Izvajalno okolje mikrostoritev 60
5.3 Orkestracija vsebnikov v arhitekturi mikrostoritev 65

6 Izboljšan model skaliranja prek metrik odpornosti na napake in validacija 73
6.1 Samodejno horizontalno skaliranje prek metrik odpornosti na napake 74
6.2 Rezultati in diskusija 88

7 Zaključek 97

A Primer konfiguracije objekta postavitve v orkestratorju vsebnikov Kubernetes prek dokumenta YAML 101

Seznam uporabljenih kratic

API	Application Programming Interface
AWS	Amazon Web Services
CDI	Contexts and Dependency Injection
CLI	Command Line Interface
CNA	Cloud-Native Application
CORS	Cross-Origin Resource Sharing
CPU	Central Processing Unit
CRUD	Create, Read, Update, Delete
DB	Database
EE	Enterprise Edition
EJB	Enterprise Java Beans
GCE	Google Compute Engine
GKE	Google Container Engine
HPA	Horizontal Pod Autoscaler
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
JAR	Java Archive
JAX-RS	Java API for RESTful Web Services
JAX-WS	Java API for XML Web Services
JPA	Java Persistence API
JSF	Java Server Faces
JSON	JavaScript Object Notation
JSON-P	JSON Processing
JSP	Java Server Pages
JTA	Java Transaction API

JWT	JSON Web Token
LB	Load Balancer
LXC	Linux Containers
OIDC	OpenID Connect
OSS	Open Source Software
PaaS	Platform as a Service
POM	Project Object Model
RAM	Random Access Memory
REST	Representational State Transfer
RPS	Requests per Second
STONITH	Shoot the Other Node in the Head
UFS	Union File System
UID	Unique Identifier
VM	Virtual Machine
vCPU	Virtual Central Processing Unit
YAML	YAML Ain't Markup Language

Povzetek

Naslov: Prožna oblačna arhitektura aplikacij z uporabo prekinjevalcev toka

Naslovili smo mehanizme odpornosti na napake v arhitekturi mikrostoritev oblačnih aplikacij. Podrobno smo proučili vzorce odpornosti na napake vključno s prekinjevalci toka ter analizirali, kako izboljšujejo odpornost ter odzivnost sistema. Zasnovali in razvili smo rešitev za uporabo vzorcev odpornosti na napake v javanskih mikrostoritvah. Podrobno smo proučili vpliv odpornosti na napake na skalabilnost in elastičnost oblačnih aplikacij in analizirali obstoječe pristope. Zasnovali smo izboljšano rešitev za samodejno skaliranje na osnovi upoštevanja metrik odpornosti na napake, ki je v primerjavi s pristopom HPA za 45 % izboljšala odstopanja od referenčnih optimalnih vrednosti.

Ključne besede

oblačna aplikacija, prekinjevalec toka, odpornost na napake, samodejno horizontalno skaliranje

Abstract

Title: Resilient cloud-native application architecture using circuit-breakers

We have addressed fault tolerance mechanisms in microservice architecture for cloud-native applications. We have examined the fault tolerance patterns including circuit breakers and analyzed how they improve the system resilience and responsiveness. We have designed and developed a solution for fault tolerance patterns in Java microservices. We studied the impact of fault tolerance on the scalability and elasticity of cloud-native applications and analyzed existing approaches. We have designed an improved scaling solution based on fault tolerance metrics, which improved the deviations from the optimal reference values compared to the HPA approach by 45 %.

Keywords

cloud-native application, circuit-breaker, fault tolerance, horizontal autoscaling

Poglavje 1

Uvod

1.1 Motivacija

Uveljavitev računalništva v oblakih (ang. *cloud computing*) je v zadnjih letih privedla do velikega napredka v arhitekturah oblačnih aplikacij. Arhitektura mikrostoritev (ang. *microservice architecture*) odpravlja pomanjkljivosti arhitekture monolitnih aplikacij in v polni meri izkorišča prednosti, ki jih prinaša računalništvo v oblaku. Monolitna arhitektura (ang. *monolithic architecture*) predstavlja aplikacijo, ki je sestavljena iz ene postavitvene enote (ang. *deploy unit*) in v kateri je vsebovana celotna aplikacijska logika [1, 2].

Arhitektura mikrostoritev uvaja porazdeljen sistem večjega števila manjših aplikacij ali mikrostoritev, kjer vsaka predstavlja svojo postavitveno enoto. Majhnost postavitvenih enot poenostavlja aplikacijsko logiko, omogoča lažjo razumljivost, obvladljivost kode ter hitrejše spremembe in povečuje skalabilnost na nivoju posamezne mikrostoritve [1]. V današnjem poslovnem svetu, kjer so hitre spremembe, visoka razpoložljivost in možnost hitrega prilagajanja ključnega pomena, je arhitektura mikrostoritev vse bolj pogosto uporabljen pristop v večini modernih aplikacij. Arhitektura s seboj prinaša številne izzive na področju izvajanja v porazdeljenih sistemih [3].

V porazdeljenem sistemu poteka komunikacija med šibko sklopljenimi (ang. *loosely coupled*) komponentami prek omrežnih komunikacijskih pro-

tokolov. Način komunikacije se močno razlikuje od komunikacije med komponentami v monolitnih aplikacijah. V monolitnih aplikacijah komunikacija večinoma poteka znotraj istega procesa prek skupnega pomnilnika in velja za hitro ter zanesljivo. Omrežne komunikacije med komponentami v arhitekturi mikrostoritev prinašajo daljše odzivne čase in znižujejo stopnjo zanesljivosti komunikacije zaradi mogočih odpovedi in napak v omrežni komunikaciji. Teh ni mogoče v celoti preprečiti in je treba z njimi ustrezno upravljati. Potrebni so mehanizmi, kot je prekinjevalec toka (ang. *circuit-breaker*), ki zagotavljajo odpornost posamezne mikrostoritve na tovrstne napake in s tem zagotavljajo stabilnost, odpornost in prožnost celotne oblačne aplikacije [1, 4, 5].

Netflix OSS (*Open Source Software*) predstavlja skupino pomembnih odprtokodnih rešitev na področju oblačnih arhitektur in mikrostoritev. Mednje spada orodje Hystrix [6], ki velja za eno najbolj pogosto uporabljenih rešitev na področju odpornosti na napake. Orodje Hystrix spremlja izvajanje operacij na oddaljeno komponento. Ob doseganju pogojev neuspešnosti začasno prepreči ponovno uporabo operacije za določen čakalni čas in na ta način zaustavi širjenje napake skozi sistem mikrostoritev [7]. Pri spremljanju izvajanja odvisnih operacij Hystrix poseduje velike količine podatkov o dostopnosti virov [8].

Dela [5, 9] naslavljajo problem samoupravljalne arhitekture oblačne aplikacije. Opisujejo pristop k zagotavljanju dinamičnega horizontalnega skaliranja (ang. *horizontal scaling*) aplikacije glede na obremenjenost komponent. Vsaka mikrostoritev predstavlja gručo (ang. *cluster*) več primerkov. V gruči se določi primerek z vodilno vlogo, ki na podlagi periodičnih poizvedb o stanju ostalih primerkov določa horizontalno skaliranje znotraj gruče. Avtorji omenijo možnost uporabe prekinjevalcev toka kot učinkovit način za zagotavljanje prožnosti, kljub temu pa prekinjevalci toka ne predstavljajo pomembnejše vloge v njihovi rešitvi.

Delo [10] predstavlja tri primere umestitve tehnologije prekinjevalcev toka v arhitekturo aplikacij na relaciji med odjemalcem storitve in samo storitvijo. Predvideva tri pristope: implementacijo prekinjevalcev toka na nivoju

odjemalca, implementacijo prekinjevalcev toka na nivoju storitve in uporaba posredniške (ang. *proxy*) storitve s prekinjevalci toka. Pristop s posredniško storitvijo je predstavljen kot najnaprednejši, a predstavlja ozko grlo v arhitekturi. Skupaj s prehodom API in registrom storitev se lahko problem ozkega grla reši.

1.2 Cilji

Prvi cilj magistrske naloge je analizirati in proučiti vzorec prekinjevalca toka ter ostalih vzorcev odpornosti na napake pri različnih načinih komunikacije med mikrostoritvami. Pregledali bomo obstoječe rešitve na področju razvoja mikrostoritev in orodij za uporabo omenjenih vzorcev. Za čim boljšo zasnovo rešitve želimo identificirati dobre prakse in načine uporabe pri že obstoječih rešitvah.

Drugi cilj je umestitev, zasnova ter implementacija rešitve za uporabo vzorcev odpornosti na napake za mikrostoritve v programskem jeziku Java. Omogočiti želimo enostavno uporabo in doseči ustrezno umestitev vzorcev znotraj same mikrostoritve. Ker je prek vzorcev odpornosti na napake na voljo velika količina podatkov o dostopnosti oddaljenih virov, je smiselno omogočiti uporabo metrik, ki se generirajo pri izvajanju zahtev ob uporabi vzorcev.

Tretji cilj magistrske naloge je zasnova in implementacija izboljšanega modela za samodejno skaliranje mikrostoritev prek metrik vzorcev odpornosti na napake. Proučili bomo vpliv odpornosti na napake na skalabilnost in fleksibilnost oblačne aplikacije. Vzorce bomo v arhitekturo oblačne aplikacije umestili kot centralni element ter na ta način izboljšali tako odpornost in odzivnost sistema kot tudi fleksibilnost in elastičnost. Zasnovano rešitev samodejnega horizontalnega skaliranja prek metrik odpornosti na napake bomo ovrednotili in primerjali s pristopom k samodejnemu horizontalnemu skaliranju v orodju Kubernetes prek odstopanja od referenčnih optimalnih vrednosti. Primerjavo bomo opravili na vzorčni aplikaciji, ki smo jo razvili v okviru

magistrske naloge.

1.3 Struktura

Na začetku bomo v poglavju 2 podrobneje predstavili arhitekturo oblačnih aplikacij in arhitekturo mikrostoritev. Pogledali bomo ključne koncepte v arhitekturi mikrostoritev in zakaj se potrebujejo. Sledil bo pregled ogrodij za razvoj mikrostoritev. Na koncu poglavja bomo definirali vrste komunikacij med mikrostoritvami in opredelili, zakaj so vzorci odpornosti na napake v komunikaciji potrebni.

V poglavju 3 bomo proučili vzorce odpornosti na napake v okviru stabilnosti ob napakah. Pregledali bomo nekaj obstoječih orodij, ki implementirajo vzorce odpornosti na napake v programskem jeziku Java. Spoznali se bomo tudi s specifikacijo, ki se razvija na tem področju.

Zasnovo in implementacijo rešitve za odpornosti na napake za mikrostoritve v Javi bomo predstavili v poglavju 4. Opisali bomo podatke, ki se generirajo pri uporabi vzorcev in jih izpostavili prek metrik. Uporabo rešitve odpornosti na napake bomo predstavili na primeru.

V poglavju 5 bomo predstavili področje skalabilnosti mikrostoritev ter načine izvajanja v oblačnih aplikacijah. Spoznali se bomo z orkestracijo mikrostoritev in pristopi k skaliranju mikrostoritev.

V poglavju 6 bomo predstavili izboljšan model samodejnega skaliranja mikrostoritev prek vzorcev odpornosti na napake. Vzorce bomo umestili v arhitekturo oblačnih aplikacij in opisali delovanje naše rešitve. Predstavili bomo testni primer in na njem opravili validacijo zasnovane rešitve.

Poglavje 2

Arhitektura oblačnih aplikacij

Oblačne aplikacije CNA (*Cloud-Native Applications*) so aplikacije, ki so arhitekturno zasnovane in implementirane tako, da v čim večji meri izkoriščajo prednosti računalništva v oblaku. Pojavile in uveljavile so se predvsem kot odgovor naraščanju števila končnih uporabnikov, hitremu prilagajanju spremembam v poslovnem svetu ter v poslovnih aplikacijah in potrebi po večji skalabilnosti. V arhitekturi oblačnih aplikacij se najpogosteje uporablja arhitektura mikrostoritev, ki predstavlja porazdeljen sistem večjega števila manjših šibko sklopljenih storitev. Arhitektura mikrostoritev v oblačnih aplikacijah odpravlja pomanjkljivosti arhitekture monolitne aplikacije [2, 1].

Monolitna arhitektura predstavlja aplikacije, ki so večinoma sestavljene iz ene postavitvene enote in znotraj nje vsebujejo celotno aplikacijsko logiko. Zaradi svoje obsežnosti in kompleksnosti so monolitne aplikacije težje obvladljive in zahtevajo globlje poznavanje aplikacijske logike za kakovosten razvoj aplikacije. Velikost postavitvene enote monolitne aplikacije vpliva na manjšo skalabilnost aplikacij. To se pozna predvsem v horizontalnem skaliranju aplikacije, ki ni dovolj učinkovito, saj ne omogoča enostavnejšega ločenega skaliranja posameznih komponent aplikacije. Oblačne aplikacije so na drugi strani največkrat sestavljene iz več postavitvenih enot, ki so enostavnejše ter bolj razumljive. To jim omogoča hitrejše prilagajanje spremembam, poenostavljeno implementacijo novih funkcionalnosti in večjo skalabilnost.

Zaradi manjših postavitvenih enot in delovanja komponent kot ločene samostojne (ang. *standalone*) aplikacije, je arhitekturo mikrostoritev mogoče bolj učinkovito horizontalno skalirati. Z izvajanjem v oblakih pridobijo oblačne aplikacije visoko razpoložljivost in prožnost [4, 1].

Na začetku bomo v poglavju 2.1 opisali arhitekturo oblačnih aplikacij. V poglavju 2.2 bomo predstavili nadgradnjo oblačnih aplikacij z arhitekturo mikrostoritev ter nato v poglavju 2.3 pogledali nekatere ključne koncepte arhitekture mikrostoritev. V poglavju 2.4 bo sledil pregled orodij, ki omogočajo razvoj mikrostoritev v programskem jeziku Java. Na koncu bomo v poglavju 2.5 predstavili vzorce komunikacije med mikrostoritvami in opisali potrebo po prekinjevalcih toka in ostalih vzorcih na področju odpornosti na napake v različnih vzorcih komunikacije.

2.1 Kratek pregled arhitekture oblačnih aplikacij

V povezavi z arhitekturo oblačnih aplikacij se pogosto omenja zbirko vzorcev 12-faktorske aplikacije (ang. *12-factor app*), ki so jo definirali inženirji podjetja Heroku [11]. Vzorci opisujejo smernice razvoja oblačnih aplikacij za izvajanje v računalniških oblakih. Definirani vzorci so naslednji:

- **Baza izvorne kode (ang. *codebase*)** — aplikacija uporablja izvorno kodo v enotni bazi ne glede na končno okolje izvajanja in omogoča sledljivost kode prek sistema za nadzor izvorne kode (ang. *version control system*).
- **Odvisnosti (ang. *dependencies*)** — aplikacija eksplicitno določa odvisnosti prek ustreznih orodij za upravljanje z odvisnostmi.
- **Konfiguracija (ang. *config*)** — vsaka konfiguracija, ki določa delovanje aplikacije in se razlikuje od preostalih postavitvenih okolij (razvojno, testno, produkcijsko), se določi prek okoljskih spremenljivk.

- **Podporne storitve (ang. *backing services*)** — podporne storitve, kot so med drugim podatkovne baze, se uporabljajo oz. povezujejo kot zunanje storitve in so lahko souporabljene s strani več aplikacij.

- **Graditev, izdaja, izvajanje (ang. *build, release, run*)** — graditev aplikacije, njena objava v različnih postavitvenih okoljih in izvajanje aplikacije v kombinacijah z različnimi pripadajočimi konfiguracijami so strogo ločeni procesi.

- **Procesi (ang. *processes*)** — aplikacija se izvaja v enem ali več procesih, ki ne ohranjajo stanja (ang. *stateless*).

- **Vezava na vrata (ang. *port binding*)** — vse storitve aplikacija izpostavlja prek vrat.

- **Sočasnost (ang. *concurrency*)** — aplikacija se lahko sočasno izvaja v več procesih vzporedno.

- **Odstranljivost (ang. *disposability*)** — aplikacija se zažene hitro in ustavi elegantno.

- **Testno produkcijska parnost (ang. *dev/prod parity*)** — aplikacija teži k čim večji podobnosti v razvojnem, testnem in produkcijskem postavitvenem okolju.

- **Dnevniški zapisi (ang. *logs*)** — beleženje dnevniških zapisov v aplikacijah se obravnava kot tok dogodkov, kar omogoča zbiranje, agregiranje, indeksiranje in analiziranje dogodkov prek zunanjih storitev za upravljanje z dnevniškimi zapisi.

- **Administratorski proces (ang. *admin process*)** — administratorske in upravljalne naloge se izvaja kot enkraten proces v identičnem okolju, v katerem se izvaja tudi navadni dolgo trajajoči proces aplikacije.

Omenjeni koncepti omogočajo čim bolj optimalno izvajanje aplikacij v računalniškem oblaku. Oblaki ponujajo teoretično neomejene računske vire, ki se skupaj z oblačno aplikacijo prilagajajo trenutnim potrebam oz. obremenitvam. Oblačna aplikacija se je sposobna hitro prilagajati spremembam. Vzorca sočasnosti in odstranljivosti sta glavna faktorja, ki omogočata prožnost arhitekture oblačnih aplikacij. Sočasnost se navezuje na možnost hkratnega izvajanja aplikacije v več procesih. Predvsem pri arhitekturi monolitnih aplikacij kot tudi v zametkih oblačnih arhitektur se je sočasnost navezovala na nitno (ang. *thread*) izvajanje in vertikalno skaliranje aplikacij. Ko je aplikacija dosegla obremenitvene meje, so se ji dodelili dodatni računski viri. Aplikacija se je povečala v vertikalni smeri. Oblačna arhitektura je uvedla skaliranje navzven (ang. *scale out*) ali horizontalno skaliranje. Tako skaliranje namesto večanja aplikacije poveča število primerkov aplikacije [3, 12].

Vzorec sočasnosti se dobro dopolnjuje z vzorcem odstranljivosti. Če v aplikaciji pride do napake, se zaradi odstranljivosti primerek aplikacije enostavno odstrani, ne da bi pri tem povzročil dodatne kritične napake na povezanih storitvah, odvisnostih ter podpornih storitvah. Ker odstranljivost predvideva tudi hitri zagon aplikacije, se novi nadomestni primerek vzpostavi hitro brez večjega vpliva na sistem. Če v času zagona nadomestnega primerka obstaja delujoč dodaten primerek aplikacije, je vpliv na delovanje sistema minimalen oz. skoraj ničen. Vzorec procesov, ki ne ohranjajo stanja, omogoča, da je izvedba sočasnosti in odstranljivosti v oblačnih aplikacijah uporabnikom sistema navzven nevidna oz. ne vpliva na uporabniško izkušnjo [3, 11, 12].

Opisani vzorci omogočajo skalabilno in dinamično arhitekturo oblačnih aplikacij. Z vpeljavo arhitekture mikrostoritev se je oblačne aplikacije povzdignilo na novo raven, ki vpeljuje veliko število novih konceptov in pristopov. Arhitektura oblačnih aplikacij je postala porazdeljena in veliko bolj dinamična, močno pa so se povečale možnosti skalabilnosti.

2.2 Arhitektura mikrostoritev

Arhitektura mikrostoritev predstavlja arhitekturni pristop k razvoju in izvajanju oblačnih aplikacij. Oblačna aplikacija, ki uporablja arhitekturo mikrostoritev, uvaja porazdeljen sistem množice manjših enot oz. (mikro)storitev, ki so šibko sklopljene. Vsaka mikrostoritev deluje kot samostojna 12-faktorska aplikacija in predstavlja svojo postavitveno enoto. Zaradi majhnosti postavitvenih enot je aplikacijska logika mikrostoritev lažje razumljiva ter obvladljiva, omogoča hitrejše spremembe, povečuje skalabilnost na nivoju posamezne mikrostoritve, omogoča hitrejšo odstranljivost, graditev ter izdajanje novih različic in omogoča gradnjo vsake posamezne mikrostoritve na različnih tehnologijah [1, 13].

Arhitektura mikrostoritev odpravlja pomanjkljivosti arhitekture monolitnih aplikacij. V uvodu poglavja smo opisali prednosti manjših postavitvenih enot mikrostoritev v oblačnih aplikacijah v primerjavi z veliko postavitveno enoto monolitne aplikacije. Zaradi manjše postavitvene enote mikrostoritev se postopek gradnje, testiranja, objavljanja in izvajanja poenostavi in pospeši, kar olajša tudi samodejne postopke zvezne gradnje (ang. *continuous build*), zvezne integracije (ang. *continuous integration*) ter zvezne dostave (ang. *continuous delivery*) posamezne komponente v arhitekturi [14, 15]. Arhitektura mikrostoritev izboljšuje skalabilnost ter z njo predvsem horizontalno skaliranje, ki je veliko bolj učinkovito zaradi hitrejšega oblikovanja ter odstranjevanja primerkov mikrostoritev in možnosti skaliranja glede na obremenjenost različnih komponent oz. mikrostoritev v aplikaciji [13, 4].

Ob pogledu na celoten porazdeljen sistem mikrostoritev se določene stvari otežijo. Nadzorovanje mikrostoritev v končnem postavitvenem okolju sistema mikrostoritev je bolj težaven proces. Z oblačnimi aplikacijami, še dodatno pa z mikrostoritvami, je posledično izrednega pomena postalo sodelovanje med razvijalci programske opreme in operaterji, ki nadzirajo in upravljajo z operacijami v oblaku. Omenjeno sodelovanje je privedlo do novega procesa v razvoju programske opreme imenovanega razvojne operacije (ang. *DevOps*). Proces skrbi za zveznost v postopku objavljanja in konstantnega dostavljanja

novih verzij. V procesu je treba poskrbeti za podporno infrastrukturo v porazdeljenih sistemih, ki je nujno potrebna za obvladovanje velikega števila primerkov mikrostoritev [3, 2, 14].

Mikrostoritve so različno medsebojno odvisne in pri povezovanju intenzivno komunicirajo s pomočjo komunikacijskih protokolov, ki delujejo prek omrežja. Pri tem se pogosto uporablja lahkotne (ang. *lightweight*) komunikacijske mehanizme in protokole, kot je npr. HTTP (*Hypertext Transfer Protocol*) prek vmesnikov REST (*Representational State Transfer*). Način komuniciranja se v porazdeljenem sistemu mikrostoritev izrazito razlikuje od komuniciranja med komponentami v monolitnih aplikacijah, kjer si komponente delijo lokalni pomnilnik, proces in ostale vire. Omrežne komunikacije v primerjavi z lokalno komunikacijo veljajo za nezanesljiv način komunikacije, velika razlika pa nastane tudi v odzivnih časih, ki so v lokalnih komunikacijah neprimerljivo nižji. V mikrostoritvah zato prihaja do napak, ki niso predvidljive in so posledica nezanesljive omrežne komunikacije v porazdeljenih sistemih in/ali njihovih visokih odzivnih časov [4]. Neodpornost na napake v komunikacijah lahko privede do kaskadnega širjenja napak, ki povzroča nedelovanje več storitev ali v najslabšem primeru nedelovanje celotne aplikacije zgolj zaradi napake v komunikaciji med dvema mikrostoritvama. Dinamičnost porazdeljenega sistema mikrostoritev, kjer se primerki posamezne storitve stalno oblikujejo in odstranjujejo, in izvajalno okolje, ki je tipično zelo omejeno s porabo CPU (*Central Processing Unit*) ter količino pomnilnika RAM (*Random Access Memory*), dodatno pripomoreta k napakam v klicih oddaljenih storitev [1].

Pristop k upravljanju z napakami se v sistemu mikrostoritev zaradi načina komunikacije med komponentami razlikuje od standardnega pristopa v monolitnih aplikacijah. Pri standardnem pristopu k napakam poskušamo napake preprečevati. Pri obravnavanju oddaljenih klicev na storitve v arhitekturi mikrostoritev pa uporabimo pristop, kjer pričakujemo, da bo do napak prišlo in jih zato ne poskušamo preprečiti, temveč se nanje čim bolje odzvati. Za ustrezno ravnanje z napakami so zato potrebni mehanizmi, ki povečujejo od-

pornost na napake tako mikrostoritve kot celotne aplikacije [4, 3]. Odpornost na napake je zato eden izmed ključnih konceptov arhitekture mikrostoritev in glavno vodilo magistrske naloge.

Poleg odpornosti na napake se je za lažje upravljanje, stabilnejše in bolj učinkovito delovanje mikrostoritev zasnovalo veliko konceptov. Ti omogočajo dinamičnost v infrastrukturah mikrostoritev, skrbijo za registracijo in odkrivanje mikrostoritev, gručenje in konfiguracijo [15]. Koncepte arhitekture mikrostoritev bomo predstavili v naslednjem poglavju.

2.3 Ključni koncepti arhitekture mikrostoritev

Glede na načine uporabe mikrostoritev, različna postavitvena okolja in množice različnih orodij se je v arhitekturi mikrostoritev pojavilo veliko konceptov, ki poenostavljajo in omogočajo boljše delovanje mikrostoritev v porazdeljenem sistemu. V tem poglavju bomo pogledali nekaj najpogostejših in ključnih konceptov, ki so se izkazali za pomembnejše in v sklopu te magistrske naloge predstavljajo vidnejšo vlogo.

2.3.1 Konfiguracija

Vzorci 12-faktorske aplikacije v eni izmed točk opisujejo konfiguracijo. Ta predvideva, da so konfiguracije, ki določajo delovanje aplikacije in se razlikujejo pri različnih postavitvenih okoljih, podane kot okoljske spremenljivke. Z uvedbo porazdeljenih sistemov skozi arhitekturo mikrostoritev in preostalih ključnih konceptov arhitekture mikrostoritev konfiguracija prek okoljskih spremenljivk ni več dovolj fleksibilna. Konfiguracije je pogosto treba spreminjati sproti, sprememba vrednosti okoljskih spremenljivk pa večinoma zahteva ponovni zagon posamezne mikrostoritve. Če je primerkov mikrostoritev več, je treba z novimi nastavitvami ponovno zagnati vse, kar pa je prevelika izguba. Konfiguracija se zato v arhitekturah mikrostoritev največkrat določa

prek konfiguracijskih strežnikov (ang. *configuration server*). Spremembe posameznih konfiguracij se adaptirajo na vse storitve brez potrebe po ponovnih zagonih. Konfiguracijski strežniki omogočajo konfiguracijo za različna postavitvena okolja in največkrat tudi za različne različice mikrostoritev ter s tem prinašajo želeno fleksibilnost [1]. Med pogosteje uporabljene rešitve na področju konfiguracijskih strežnikov spadajo CoreOS etcd[1], HashiCorp Consul[2] in Spring Cloud Config[3].

2.3.2 Odkrivanje storitev

V porazdeljenem sistemu mikrostoritev se nahaja veliko število samostojnih aplikacij, ki predstavljajo posamezno mikrostoritev in imajo pogosto več vzporedno izvajajočih primerkov za doseganje večje zmogljivosti. V sistemu mikrostoritev lahko posamezni primerki predstavljajo različne verzije in različna postavitvena okolja pri posamezni storitvi. Spremljanje in sledenje posameznim primerkom storitev ter njihovem stanju ob neprestani dinamičnosti oblikovanja in odstranjevanja bi bilo na statičen način nemogoče doseči. Storitev se ob postavitvi registrira v namenski strežnik, ki skrbi za ohranjanje trenutnega stanja prijavljenih mikrostoritev. Pri tem se lahko registrira s specifično verzijo, postavitvenim okoljem, dostopnim vratom, in tudi s potjo (ang. *path*) za dostop. Ko želi odjemalna aplikacija dostopati do neke storitve, od namenskega strežnika zahteva dostopne podatke glede na kriterije (verzija, postavitveno okolje). Namenski strežnik poskrbi za zagotovitev ustreznega dostopnega naslova [1]. Med rešitvami za strežnik pri odkrivanju storitev se pogosto uporablja Consul, etcd in Netflix Eureka[4].

[1] `https://coreos.com/etcd/`
[2] `https://www.consul.io`
[3] `https://cloud.spring.io/spring-cloud-config/`
[4] `https://github.com/Netflix/eureka`

2.3.3 Izenačevalec bremena in robni strežnik

Izenačevalec bremena LB (*Load Balancer*) skrbi za enakomerno porazdelitev bremena med primerke, ki predstavljajo isto mikrostoritev. Razporejevalec bremena podatke o primerkih posameznih mikrostoritev pridobi prek strežnika za odkrivanje storitev in na podlagi izbirnega algoritma določi konkreten primerek, na katerega bo zahteva preusmerjena [1].

Robni strežnik (ang. *edge server*) je prav tako pomemben koncept v arhitekturi mikrostoritev. Gre za implementacijo prehoda API (ang. *API gateway*), ki skrbi za preusmerjanje zahtev iz javno izpostavljenih storitev na ustrezne primerke mikrostoritev v interni arhitekturi. Na ta način se zunanje odjemalce izolira od uporabljene notranje arhitekture. Notranje spremembe, ki se odvijajo v porazdeljenih sistemih, tako ne vplivajo na zunanje odjemalce [1].

2.3.4 Odpornost na napake

Do napak v arhitekturi mikrstoritev znotraj oblačnih aplikacij prihaja zaradi narave komunikacije med šibko sklopljenimi storitvami v porazdeljenih sistemih in se jim ni mogoče izogniti. Vzroke za napake smo opisali v poglavju 2.2. Pravilen pristop k napakam z uvedbo koncepta odpornosti na napake je zato izredno pomemben. Koncept odpornosti na napake se izvaja na več nivojih s pomočjo različnih vzorcev, med katerimi je prekinjevalec toka eden od ključnih vzorcev. Vzorce odpornosti na napake in pristope k stabilnosti ob napakah bomo podrobneje obravnavali v poglavju 3.1.2.

2.3.5 Dnevniški zapisi, metrike in preverjanje vitalnosti

Beleženje dnevniških zapisov je ena od tehnik nadzorovanja aplikacij, ki jo poznamo in uporabljamo že dolgo. V arhitekturah mikrostoritev in oblačnih aplikacijah se je to področje precej spremenilo. Odlaganje dnevniških zapisov v sistemske datoteke ni več primerno zaradi dinamičnosti mikrostoritev.

Dnevniške zapise je zato treba izvoziti v zunanje centralne sisteme. Ti zbirajo, hranijo in agregirajo dnevniške zapise iz celotne oblačne infrastrukture ter omogočajo podrobnejše in napredne vpoglede vanje [12]. Najpogostejši pristop k beleženju v oblačnih aplikacijah je z uporabo sklada Elastic Stack, ki predstavlja orodja Logstash[5], Elasticsearch[6] in Kibano[7]. Logstash skrbi za centralno zbiranje dnevniških zapisov iz različnih virov, njihovo procesiranje ter transformacijo in hranjenje v Elasticsearch [16]. Elasticsearch je porazdeljen analitični pogon za iskanje po velikih količinah podatkov [17]. Kibana je orodje za napredno vizualizacijo podatkov, pridobljenih prek orodja Elasticsearch [18]. Skupaj tvorijo uveljavljeno strukturo za nadzorovanje in pregledovanje dnevniških zapisov iz različnih virov.

Podoben problem kot pri beleženju se je v arhitekturi mikrostoritev pojavil tudi pri metrikah. Metrike nudijo vpogled v zmogljivost aplikacije in sistemov, na katerih aplikacije tečejo. Na podlagi metrik je mogoče nadzorovanje (ang. *monitoring*) in obveščanje (ang. *alerting*) ob izrednih situacijah. Tudi iz dnevniških zapisov je mogoče pridobiti poglede na zmogljivosti sistema. Če primerjamo metrike in dnevniške zapise, ugotovimo, da je za pridobitev podatkov iz dnevniških zapisov potrebno veliko dodatne obdelave podatkov, medtem ko so metrike bolj neposredne. Na drugi strani dnevniški zapisi ponujajo bolj podroben vpogled v dogajanje v aplikaciji, kar je zelo koristno pri odpravljanju napak [12, 19]. Metrike lahko, podobno kot dnevniške zapise, uvozimo v sklad Elastic Stack. Pri zbiranju metrik so sicer bolj razširjena druga orodja, med najbolj znanimi sta Prometheus[8] in Graphite[9]. Pri pridobivanju metrik v grobem ločimo dva različna pristopa, in sicer povlečni (ang. *pull*) ter potisni (ang. *push*). Pri potisnem pristopu nadzirane storitve potiskajo metrike v centralno storitev za metrike. Povlečni pristop deluje ravno obratno. V tem primeru storitev za metrike pridobiva oz. povprašuje po

[5] `https://www.elastic.co/products/logstash`
[6] `https://www.elastic.co/products/elasticsearch`
[7] `https://www.elastic.co/products/kibana`
[8] `https://prometheus.io`
[9] `https://graphiteapp.org`

metrikah nadzirane storitve. Rešitvi Graphite in Prometheus se razlikujeta ravno v tem, da Graphite uporablja potisni, Prometheus pa povlečni pristop [20, 21].

Preverjanje vitalnosti (ang. *health check*) je dodaten mehanizem za preverjanje stanja v mikrostoritvi. Včasih se lahko zgodi, da je primerek mikrostoritve neodziven na zahteve in se kljub temu še vedno izvaja. Tovrstne primere je treba zaznati. Metrike so eden od pristopov, ki pa ni dovolj hiter. V tem času primerek mikrostoritve ne sme prejemati zahtev. Odkrivanje storitev zato v številnih implementacijah uporablja periodično preverjanje vitalnosti posameznih primerkov mikrostoritev, ki odkrivanju storitev javi aktualno stanje vitalnosti. Preverjanje vitalnosti vsebuje osnovne podatke o stanju okolja, v katerem se storitev izvaja, vitalnosti vzpostavljenih povezav (npr. s podatkovno bazo) in morebitne dodatne podatke o stanju aplikacije. Če stanje ni vitalno, bo odkrivanje storitev poskrbelo za ustrezne akcije in ranjenemu primerku ne bo posredovalo zahtev [22].

2.4 Pregled ogrodij za razvoj mikrostoritev v programskem jeziku Java

Za razvoj mikrostoritev obstaja veliko število ogrodij, ki omogočajo implementacijo in izvajanje mikrostoritev v računalniških oblakih. Razvitih je veliko različnih komponent za podporo mikrostoritvam, med katere spadajo tudi ključni koncepti mikrostoritev. V tem delu se bomo osredotočili na ogrodja, ki so na področju obravnavane problematike na voljo za programski jezik Java. Med javanska ogrodja, ki omogočajo razvoj mikrostoritev, spadajo Spring Boot/Cloud, WildFly Swarm, KumuluzEE, Payara, Vert.x itd. Na področju razvoja mikrostoritev za programski jezik Java se je zasnovala in izoblikovala specifikacija MicroProfile, ki poskuša poenotiti način razvoja mikrostoritev. V naslednjih podpoglavjih bomo pregledali najbolj razširjena ogrodja in specifikacijo za razvoj mikrostoritev.

2.4.1 Spring Boot in Spring Cloud

Spring Boot [23] je orodje, ki ga je razvilo podjetje Pivotal Software in omogoča razvoj produkcijskih aplikacij in storitev na čim bolj preprost način. Zasnovano je s ciljem, da je treba vložiti minimalno količino začetnega truda za implementacijo in zagon začetne aplikacije. Spring Boot je samostojna aplikacija Java in omogoča preprosto vključitev javanskega strežnika in vsebnika za servlet, kot sta Jetty in Tomcat. Ponuja samodejno konfiguracijo glede na tip aplikacije in glede na vključene komponente. Z uporabo različnih prednastavljenih začetnih paketov omogoča vključitev prednastavljenih odvisnosti. Izbira začetnega projekta je odvisna od tipa aplikacije [24]. Paket se lahko določi prek orodij Maven[10] in Gradle[11], ki veljata za najbolj razširjeni orodji za upravljanje projektov programskega jezika Java.

Zagon aplikacije, razvite z orodjem Spring Boot, je zelo hiter. Velika prednost orodja Spring Boot se pokaže ob pregledu nabora projektov, ki jih vzdržuje Spring in jih lahko vključimo v aplikacijo Spring Boot. Med zanimivejšimi je projekt Spring Cloud [25]. Projekt ponuja orodja, ki omogočajo uporabo ključnih konceptov za uporabo v porazdeljenih sistemih oz. v mikrostoritvah. Pri uporabi konceptov ponujajo tako uporabo lastnih rešitev kot integracijo znanih knjižnic na različnih področjih. Med bolj znane spadajo rešitve Netflix OSS. Med pomembnejše podprte koncepte orodja Spring Cloud spadajo porazdeljena in verzionirana konfiguracija, registracija in odkrivanje storitev, usmerjanje, klici med storitvami (ang. *service to service calls*) in izenačevanje bremena.

2.4.2 WildFly Swarm

WildFly Swarm [26] ponuja inovativen pristop k pakiranju in izvajanju aplikacij Java EE (*Enterprise Edition*). Pri tem v paket vključi minimalno število odvisnosti, ki so potrebne za zagon javanskega strežnika prek arhiva

[10] `https://maven.apache.org`
[11] `https://gradle.org`

JAR (*Java Archive*). V aplikacijo se poljubno vključuje potrebne komponente. WildFly Swarm podpira vključevanje standardiziranih komponent Java EE in je kompatibilen s specifikacijo MicroProfile. Poleg komponent Java EE ponuja tudi rešitve za ključne koncepte v arhitekturi mikrostoritev v oblačnih aplikacijah, med katerimi je podpora za odpornost na napake z uporabo orodja Hystrix, odkrivanje storitev, izenačevalec bremena, beleženje dnevniških zapisov tako prek standardnega sistema orodja JBoss kot tudi izvoz v orodja Logstash in Fluentd. Podprto je tudi preverjanje vitalnosti in porazdeljeno sledenje (ang. *distributed tracing*). Pri podpori komponent standardov Java EE je smiselno omeniti podporo standardu EJB, ki je ena od ključnih komponent Jave EE v aplikacijskih strežnikih in je podprta v malokaterem ogrodju za razvoj mikrostoritev. Standard EJB se praviloma v mikrostoritvah ne uporablja, saj je bil zasnovan z namenom omogočanja strukturiranja monolitne aplikacije v komponente, ki so skalabilne, se lahko nahajajo na različnih napravah in znajo med seboj komunicirati [27]. Standard na neki način predstavlja mikrostoritve znotraj monolitne aplikacije. Ob uporabi standarda v mikrostoritvah se izniči sam namen mikrostoritev. Poleg tega standard EJB s seboj prinaša kompleksnejše tehnologije, ki so za mikrostoritve nepotrebne.

Orodje poleg bolj znanih in razširjenih knjižnic ponuja tudi veliko lastnih rešitev, marsikatera pa je kot komponenta vpeljana iz aplikacijskega strežnika WildFly. Mednje spada npr. rešitev za gručenje, imenovana JGroups, in skrbi za usklajevanje bremena med posameznimi primerki strežnikov, ki med seboj komunicirajo s pomočjo metode IP multicast. Enako velja tudi za podporo upravljalni konzoli (ang. *management console*). Z uporabo tehnologije JGroups omogoča WildFly Swarm rešitev na področju odkrivanja storitev. Poleg omenjene rešitve je omogočena tudi bolj pogosta uporaba odkrivanja storitev prek namenskega strežnika za odkrivanje in registracijo storitev Consul. V platformi WildFly Swarm je omogočena integracija avtentikacijskega strežnika Keycloak, ki ga prav tako razvija podjetje Red Hat in omogoča vključitev avtentikacije in avtorizacije v aplikacijo. WildFly Swarm podpira

nekatere rešitve platforme Spring. Vgrajena je podpora rešitvi Swagger za pogodbe v vmesnikih API.

2.4.3 KumuluzEE

KumuluzEE je lahkotno odprtokodno ogrodje, razvito specifično za implementacijo mikrostoritev s pomočjo standardnih tehnologij Java EE in velja za eno izmed prvih ogrodij na tem področju. Vključuje podporo standardnim Java EE tehnologijam, kot so CDI, EJB, JPA, JAX-RS, JSON-P, JAX-WS, JTA, Bean Validation, JSF, JSP in WebSocket. KumuluzEE je kompatibilen s specifikacijo MicroProfile. Ogrodje ima vgrajeno podporo za konfiguracijo aplikacije prek razširitve KumuluzEE Config, kot je definirano v specifikaciji MicroProfile Config. Podrobneje bomo specifikacijo MicroProfile in pripadajočo specifikacijo konfiguracije spoznali v poglavju 2.4.4. Ogrodje v primerjavi z definirano konfiguracijo v specifikaciji MicroProfile omogoča dodaten konfiguracijski vir, in sicer prek dokumenta YAML (*YAML Ain't Markup Language*).

KumuluzEE poleg standardnih tehnologij Java EE omogoča tudi uporabo različnih razširitev. Med njimi so predvsem razni koncepti, ki se uporabljajo v arhitekturi mikrostoritev. Podprti so dodatni konfiguracijski viri, in sicer z uporabo konfiguracijskih strežnikov etcd in Consul prek razširitve KumuluzEE Config. Prek omenjenih konfiguracijskih strežnikov je podprto odkrivanje storitev KumuluzEE Discovery. Nadalje KumuluzEE podpira beleženje dnevniških zapisov z ogrodjem Apache Log4j 2[12] ter JUL[13] prek razširitve KumuluzEE Logs, odpornost na napake z orodjem Hystrix prek razširitve KumuluzEE Fault Tolerance, metrike z rešitvami Graphite, Prometheus ter Logstash prek razširitve KumuluzEE Metrics in preverjanje vitalnosti prek razširitve KumuluzEE Health. Poleg ključnih konceptov, opisanih v poglavju 2.3, ogrodje dodaja tudi razširitev KumuluzEE REST za pogoste ter napre-

[12] `https://logging.apache.org/log4j/2.x/`

[13] `https://docs.oracle.com/javase/7/docs/api/java/util/logging/package-summary.html`

dne vzorce pri implementaciji storitev REST, razširitev KumuluzEE Event Streaming za pretakanje dogodkov (ang. *event streaming*), razširitev KumuluzEE CORS za podporo mehanizmu kontrole dostopa CORS (*Cross-Origin Resource Sharing*) in razširitev KumuluzEE Security za podporo integracije avtentikacije ter avtorizacije prek standarda OpenID.

2.4.4 Specifikacija MicroProfile

MicroProfile je specifikacija za poenotenje in optimizacijo standarda Java EE za uporabo v arhitekturi mikrostoritev. Projekt vodi fundacija Eclipse. Cilj projekta je poenotenje uporabe standardiziranih komponent Jave EE skozi številne platforme za mikrostoritve, ki na različne načine ponujajo uporabo teh komponent. V projektu MicroProfile se združujejo strokovnjaki s področja Jave EE in razvojnih ekip posameznih platform za mikrostoritve, ki poskušajo uskladiti uporabo komponent Jave EE, kot tudi definirati način uporabe ključnih konceptov v arhitekturi oblačnih aplikacij v platformah za razvoj mikrostoritev. Med platforme in podjetja, ki se ukvarjajo z razvojem mikrostoritev in podpirajo specifikacijo MicroProfile spadajo IBM, Red Hat, Payara, Fujitsu in KumuluzEE [28].

Trenutno aktualna verzija specifikacije MicroProfile v1.2 [29] vsebuje definicije in vključitev komponent Java EE za kontekste ter vključevanje odvisnosti CDI, definicijo ter implementacijo spletnih storitev REST, imenovano JAX-RS, vmesnika za procesiranje dokumentov JSON (*JavaScript Object Notation*), imenovanega JSON-P in pogostih anotacij. V specifikaciji MicroProfile v1.2 je poleg standardnih komponent Java EE specificiranih tudi nekaj konceptov arhitekture mikrostoritev, med katere spadajo specifikacija MicroProfile Fault Tolerance za odpornost na napake, MicroProfile Metrics za metrike, MicroProfile Health Check za preverjanje vitalnosti in MicroProfile Config za konfiguracijo.

Konfiguracijo smo že omenili pri ogrodju KumuluzEE v poglavju 2.4.3, kjer smo omenili dodatne vire oz. konfiguracijske vire, ki jih omogoča specifikacija MicroProfile Config. Ta definira tri privzete konfiguracijske izvore,

ki si sledijo v naslednjem prioritetnem vrstnem redu:

- **sistemske javanske nastavitve**,
- **okoljske spremenljivke**,
- **nastavitvene javanske datoteke** `.properties` dodane v razredni poti (ang. *classpath*) izvajalnega okolja.

Omogočeno je dodajanje lastnih izvorov konfiguracij, kjer je mogoče vključevanje poljubnih virov, kot je npr. podatkovna baza ali konfiguracijski strežnik [30].

Specifikacijo MicroProfile Fault Tolerance bomo podrobneje spoznali v naslednjem poglavju. Poleg vseh specifikacij, ki določajo način uporabe za koncepte arhitekture mikrostoritev, v specifikacijo MicroProfile spada še specifikacija MicroProfile JWT Authentication za nadzor dostopa (ang. *role based access control*) z uporabo žetonov JWT (*JSON Web Token*) prek identitetne plasti OIDC (*OpenID connect*) na protokolu OAuth.

2.4.5 Povzetek ogrodij in specifikacije

Skozi pregled ogrodij smo spoznali različne pristope in načine razvoja mikrostoritev, ki jih ponujajo predstavljena ogrodja. Moč ogrodja Spring Boot je v ekosistemu projektov Spring, med katerimi je med vidnejšimi z vidika mikrostoritev predvsem projekt Spring Cloud, ki nudi podporo konceptom arhitekture mikrostoritev. WildFly Swarm je ogrodje, ki je nastalo iz aplikacijskega strežnika WildFly in je prirejeno za razvoj in izvajanje mikrostoritev. Podpira komponente Java EE in omogoča veliko dodatnih funkcionalnosti, ki so večinoma prav tako komponente aplikacijskega strežnika WildFly in jih lahko opcijsko vključujemo v ogrodje. Med njimi so koncepti arhitekture mikrostoritev. KumuluzEE je ogrodje, ki je bilo namensko zasnovano za razvoj mikrostoritev z uporabo komponent Java EE. Opcijsko omogoča vključevanje razširitev za podporo konceptom arhitekture mikrostoritev. Ker je ogrodje

namensko zasnovano za izvajanje mikrostoritev, omogoča hiter zagon aplikacije, ne vsebuje odvečnih komponent, zmanjšuje odvečne stroške (ang. *overhead*) pri izvajanju in omogoča pretvorbo izvorne kode v arhiv JAR, ki je zaradi lahkotnosti ogrodja zelo majhen. Ogrodji WildFly Swarm in KumuluzEE podpirata specifikacijo MicroProfile, ki standardizira način implementacije mikrostoritev v programskem jeziku Java za komponente standarda Java EE.

V tej magistrski nalogi je KumuluzEE ogrodje, ki ga bomo uporabljali kot osnovno ogrodje za razvoj mikrostoritev in pri implementaciji zasnovanih rešitev za mikrostoritve v Javi. Orodje smo izbrali zaradi njegove specializacije na področje razvoja mikrostoritev in podpore specifikaciji MicroProfile, kar mu daje glavno prednost v primerjavi z ogrodji Spring Boot in WidlFly Swarm.

2.5 Vzorci komunikacije med mikrostoritvami in potreba po odpornosti na napake

Mikrostoritve so manjše aplikacije, ki predstavljajo sestavni del večje aplikacije. Za delovanje celotne aplikacije je potrebno sodelovanje med komponentami. Mikrostoritve zato med seboj intenzivno komunicirajo, sodelujejo in si izmenjujejo podatke. Komunikacijo med mikrostoritvami lahko dosežemo na zelo različne načine. Poznamo več vzorcev komunikacije, ki oblikujejo različne modele komunikacije in načine pošiljanja sporočil.

V naslednjih podpoglavjih bomo pogledali pristope h komunikaciji med mikrostoritvami in potrebo po prekinjevalcih toka in odpornosti na napake, ki se v vzorcih pojavljajo. Načine reševanja izpostavljenih problemov v vzorcih komunikacije z uporabo prekinjevalcev toka in vzorcev za odpornost na napake bomo predstavili v poglavju 3.

2.5.1 Sinhron vzorec komunikacije

Sinhron vzorec komunikacije opisuje izvajanje komunikacije z oddaljeno storitvijo, pri katerem izveden klic na odjemalcu blokira izvajanje, dokler se operacija ne zaključi. Sinhron vzorec komunikacije predstavlja najbolj osnovni model komunikacije zahteva/odgovor (ang. *request/response*). Pri modelu zahteva/odgovor odjemalec pošlje zahtevo na strežnik, od katerega pričakuje odgovor. Protokol HTTP predstavlja tipičen in najbolj razširjen primer modela zahteva/odgovor, saj ob vsaki poslani zahtevi pričakuje rezultat. Najbolj pogost način pošiljanja sporočil na oddaljeno storitev v primeru uporabe sinhronega vzorca komunikacije v modelu zahteva/odgovor je prek vmesnikov REST [31].

Uporaba prekinjevalcev toka in vzorcev odpornosti na napake je v sinhronem vzorcu komunikacije pomembna. Težava je namreč v blokiranju izvajanja do zaključka sprožene operacije. Morebitna neodzivna oddaljena storitev, na katero se izvede klic, predstavlja velik problem, saj lahko privede do neskončnega čakanja na zaključek operacije. Če je v vrsti več zaporedno povezanih mikrostoritev, ki uporabljajo sinhroni vzorec komunikacije, lahko napaka v eni komponenti ali neustrezno ravnanje ob neodzivnosti v verigi privede do kaskadnega širjenja napak. Ker klice na odjemalcu navadno izvajajo niti, to pomeni zasedenost in posledično neodzivnost niti. Z uporabo prekinjevalcev toka in vzorcev odpornosti na napake se izvajanje sinhronega vzorca komunikacije največkrat časovno omeji. Uvede se tudi dodatne mehanizme, ki zagotavljajo hiter odziv ob nedosegljivosti ali pričakovanem napačnem delovanju oddaljene storitve. Omejitev števila vzporednih izvajanj je prav tako pogost pristop k omejevanju škode, ki jo lahko povzroči nepravilno delovanje oddaljene storitve ob sinhronem vzorcu komunikacije [31, 32, 33].

2.5.2 Asinhron vzorec komunikacije

Pri asinhronem vzorcu komunikacije odjemalec izvede klic na oddaljeno zahtevo, pri čimer ne čaka na dokončanje operacije. V nekaterih primerih od-

jemalca ne zanima niti končna uspešnost operacije. Z asinhronim vzorcem komunikacije se pogosteje izvaja enosmerni model komunikacije, doseže pa se lahko tudi model zahteva/odgovor.

Enosmerni model v asinhroni komunikaciji je najbolj preprost. Pogosto se model imenuje izvedi in pozabi (ang. *fire and forget*). Model komunikacije se lahko uporabi za izvajanje dolgo trajajočih procesov, kjer bi bilo nesmiselno puščati komunikacijske kanale odprte za dalj časa. Enosmerni model se uporabi predvsem v primerih, ko zanesljivost prenosa sporočila ni kritičnega pomena in je ključna predvsem hitra izvedba pošiljanja zahteve in nadaljevanje izvajanja. Uporaba prekinjevalcev toka in večine preostalih vzorcev odpornosti na napake v primeru uporabe enosmernega modela pri asinhronem vzorcu komunikacije ni smiselna, saj glavni namen tovrstne komunikacije ni zanesljivost in odpornost delovanja, temveč hitra izvedba [31, 32].

Vzorec asinhrone komunikacije se lahko uporabi tudi pri modelu zahteva/odgovor. Težji del nastopi pri odgovoru na zahtevo, kjer so mogoči različni pristopi k doseganju asinhronega vzorca. Način doseganja asinhronega odgovora je lahko zelo različen in je odvisen od uporabljene tehnologije. Največkrat se uporabi pristop z izvajanjem zahteve v ločeni niti, ki ne blokira glavne niti. Ob prejetem odgovoru se izvajanje vrne v glavno nit. Zelo podoben je tudi pristop z uporabo bazena povezav (ang. *connection pool*). Doseganje vzorca v nekaterih tehnologijah je omogočen z uporabo povratnih funkcij (ang. *callback*), ki se izvedejo ob prejemu odgovora. Obstajajo tudi pristopi in orodja, ki zakrijejo uporabljeno tehnologijo za doseganje asinhronega vzorca prek modela zahteva/odgovor. Primer je reaktivno programiranje (ang. *reactive programming*), ki ponuja zelo podobno uporabo pristopa za različne programske jezike prek izvedbe funkcij ob prejetem dogodku, ki predstavlja asinhron odgovor na zahtevo [31, 32, 33].

V uporabi modela zahteva/odgovor pri vzorcu asinhrone komunikacije je uporaba prekinjevalcev toka in vzorcev odpornosti na napake enako smiselna kot v sinhronem vzorcu. V asinhronem vzorcu je še bolj smiselna uporaba

vzorca odpornosti na napake za omejevanje maksimalnih sočasnih izvedb določenih klicev na oddaljeno komponento ob uporabi ločenih niti za izvajanje, s čimer se omeji poraba virov.

Vedno bolj pogost pristop k asinhronemu vzorcu komunikacije v mikrostoritvah je z uporabo dogodkovnih kanalov. Razlike v tehnologijah, ki omogočajo uporabo dogodkovnih kanalov, so precejšne. Model komunikacije, ki ga prinašajo, se prav tako zelo razlikuje od uporabljene tehnologije. Vsem je skupni imenovalec posrednik, kamor proizvajalec (ang. *producer*) odda sporočilo, porabnik (ang. *consumer*) pa ga prevzame. Pri tem torej ni nujno potrebno, da je prejemnik ob oddajanju sporočila prisoten in obratno. Z vidika uporabe prekinjevalcev toka in vzorcev odpornosti na napake je ključna predvsem komunikacija s posrednikom. Ta lahko uporablja tako enosmerni model kot model odgovor/zahteva. Uporaba odpornosti na napake je tudi v tem načinu komunikacije zelo pomembna. Kljub temu da so rešitve zasnovane za delovanje v porazdeljenih sistemih in posredniki veljajo za zelo zanesljive, se na to ne moremo zanašati. Nekatere rešitve za dogodkovne kanale pri komuniciranju interno že uporabljajo principe odpornosti na napake, ne pa vse. Implementacijo odpornosti na napake je zato treba, če niso že implementirane s strani odjemalcev, dodati [32, 34].

Poglavje 3

Odpornost na napake

V arhitekturi mikrostoritev je odpornost na napake eden izmed ključnih konceptov. Velika količina komunikacije v porazdeljenih sistemih mikrostoritev v primerjavi s komunikacijo v monolitnih aplikacijah poteka prek nezanesljivih kanalov, kjer prihaja do napak, ki se jim ni mogoče v celoti izogniti. Pristop k tovrstnim napakam se razlikuje od standardnega pristopa, kjer napake preprečujemo, temveč k njim pristopimo tako, da napake pričakujemo ter se poskušamo nanje čim bolje odzvati. Koncept odpornosti na napake v mikrostoritvah zato uvaja mehanizme oz. vzorce, ki z različnimi rešitvami in pristopi poskušajo reševati omenjene težave.

V poglavju 3.1 bomo predstavili področje stabilnosti ob napakah. V njem opisujemo različne vzorce znotraj koncepta odpornosti na napake, ki se uporabljajo pri upravljanju z napakami v mikrostoritvah. Predstavili bomo naloge prekinjevalcev toka in njihovo delovanje v sodelovanju z ostalimi vzorci. V poglavju 3.2 sledi pregled in primerjava javanskih orodij, ki implementirajo prekinjevalce toka ter ostale vzorce odpornosti na napake.

3.1 Stabilnost ob napakah

V arhitekturi mikrostoritev in porazdeljenih sistemih se spremembe odvijajo zelo hitro. Posamezne storitve med seboj komunicirajo, se povezujejo, pri-

merki storitev se pojavljajo in izginjajo, življenjski cikel in življenjska doba posameznih komponent sta nepredvidljiva. V tako dinamičnem in nepredvidljivem sistemu je treba poskrbeti za stabilnost v primeru napak [3]. V arhitekturi monolitnih aplikacij se veliko energije, časa in denarja vlaga v odpravljanje in preprečevanje napak. Vse napake, do katerih lahko pride v arhitekturi mikrostoritev, je nemogoče napovedati in predvideti. Te se lahko pojavijo zaradi izpada komunikacije, ki je posledica dinamičnosti omrežja v porazdeljenem sistemu mikrostoritve, ali pa zaradi napake v izvajalnem okolju mikrostoritve, ki je zaradi majhnosti aplikacije pogosto zelo omejeno z viri CPU in pomnilnikom RAM. Mikrostoritve je zato treba načrtovati za napake [4].

Področje stabilnosti in varnost ob napakah lahko razdelimo v več kategorij — vidnost napak, izolacija ter odpornost na napake in samodejno okrevanje.

3.1.1 Vidnost napak

Vidnost napak je zelo pomembna točka stabilnosti pri napakah, saj omogoča, da napako prej opazimo. V mikrostoritvah obstaja vrsta orodij, ki omogočajo vidnost napak in določanje delov sistema, ki so ob določeni napaki prizadeti. Metrike in dnevniški zapisi so pogosti vzorci, ki smo jih omenjali v poglavju 2.3.5. Bogat nabor metrik omogoča lokalizacijo napake znotraj arhitekture mikrostoritev in identifikacijo prizadetih komponent. Vse to je mogoče z dobro zasnovanim in vzpostavljenim sistemom opazovanja metrik, ki zazna odstopanja od običajnih vrednosti oz. presežek vrednosti čez alarmantno mejo [3]. Vidnost napak je osnova za soočanje z napakami v mikrostoritvah in uveljavljanjem nadaljnjih vzorcev.

3.1.2 Izolacija in odpornost na napake

Izolacija napak skrbi za omejevanje tveganja, ki se pojavi z napako. Doseg napake je smiselno čim bolj omejiti. V arhitekturi mikrostoritev je idealno napako omejiti le na eno samo mikrostoritev oz. na primerek mikrostoritve.

Doseganje optimalne omejitve mikrostoritev pa je mogoče le v navezavi z odpornostjo na napake [3].

Odpornost na napake preprečuje kaskadno širjenje napak med mikrostoritvami. Brez uvedbe odpornosti na napake v mikrostoritvah lahko odpoved ene od mikrostoritev povzroči odpoved več mikrostoritev, ki so od nedelujoče mikrostoritve odvisne [3]. V najslabšem primeru to lahko pomeni izpad oz. nedelovanje celotnega sistema. Eden najbolj prepoznavnih načinov za preprečevanje širjenja kaskadnih napak je prekinjevalec toka, ni pa edini. Poleg prekinjevalca toka poznamo še časovnik (ang. *timeout*), ponovno poskušanje (ang. *retry*), hitri neuspeh (ang. *fail fast*), nadomestni mehanizem (ang. *fallback*) in pregrade (ang. *bulkheads*).

Časovnik

Časovnik doseže stabilnosti ob napakah z omejevanjem časa za izvedbo določene akcije ali funkcije. S klicem na oddaljeno komponento z uporabo časovnika se preprečuje predolgo čakanje na odziv ob morebitni neodzivnosti klicane komponente. Časovnik je močno povezan s prekinjevalcem toka, hitrim neuspehom in nadomestnim mehanizmom [3]. Velikokrat se časovnik uporablja skupaj z vzorcem ponovnega poskušanja. Kljub smiselnosti časovnika pri klicih zunanjih odvisnosti uporaba zgolj časovnika v večini primerov ni smiselna [35].

Ponovno poskušanje

Ponovno poskušanje se največkrat uporablja skupaj s časovnikom. Vzorca skupaj delujeta tako, da po izteku časovnika za izvedbo neke akcije nastopi ponovno poskušanje, ki poskrbi za ponovno izvedbo iste akcije. Pred ponovno izvedbo akcije se doda dodaten konstanten časovni zamik (ang. *delay*), ki mu velikokrat prištejemo še nekaj naključnega odstopanja oz. trepetanja (ang. *jitter*).

Ponovno poskušanje ni nujno najboljši pristop. Če ima klicana komponenta večji problem, se bo zelo verjetno čas iztekel tudi ob ponovnem

poskušanju. Verjetnost za ponovni iztek časovnika je še toliko večji, ko ponovno poskušanje izvajamo z nizkim časovnim zamikom. Če je komponenta neodzivna, bo skupni čakalni čas za odziv primerno večji ob ponovnem poskušanju, kar pa je nasprotno od želenega doprinosa stabilnosti ob napakah. Večja smiselnost uporabe ponovnega poskušanja je pri izvajanju nalog (ang. *job*), ki niso časovno kritične. Ob neuspešni izvedbi se naloga postavi v čakalno vrsto za daljši čas, preden se poskusi s ponovno izvedbo akcije [35].

Hitri neuspeh

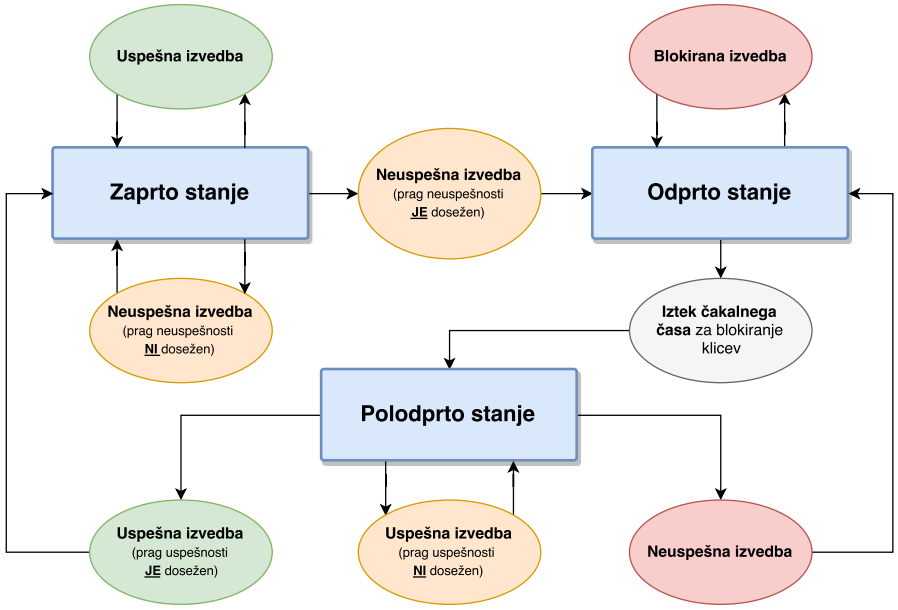
Vzorec hitrega neuspeha se uporablja v primerih, ko aplikacija vnaprej pričakuje napako pri izvajanju določene operacije. S hitrim neuspehom aplikacija prihrani pri računskih virih, ki bi se porabljali za nepotrebno procesiranje. Na drugi strani je hkrati tako odjemalska storitev kot tudi končni odjemalec veliko bolj zadovoljen s hitrim neuspešnim odzivom kakor z dolgo trajajočim odzivom, ki je povrhu še neuspešen.

Pristop k pričakovanju napake je lahko zelo enostaven. Pri klicih na storitve, ki pričakujejo vhodne parametre, je že preverjanje parametrov lahko eno od preprostih preverjanj, kjer lahko vnaprej, pred izvedbo klica, ugotovimo morebitno napako. Zelo pogosto je hitri neuspeh uporabljen pri prekinjevalcih toka [35].

Prekinjevalec toka

Prekinjevalec toka je vzorec, ki je dobro poznan koncept s področja električnega kroga in toka. V električnem krogu prekinjevalec toka ali varovalka deluje kot samodejno stikalo, ki skrbi za varovanje električnega kroga pred poškodbami, ki ga lahko povzroči presežek električnega toka. Presežek električnega toka je tipično posledica preobremenitve električnega kroga. Z odprtjem kroga ob presežku električnega toka se preprečuje škodo na elementih, ki so povezani v električni krog.

Koncept je iz električnega kroga prenesen na področje odpornosti na napake v arhitekturi mikrostoritev. Na povezavo med dvema komponentama



Slika 3.1: Diagram prehajanja stanj v prekinjevalcu toka.

lahko gledamo kot na krog. Ob normalnem delovanju je krog zaprt, prekinjevalec toka pa v zaprtem stanju. Ko prekinjevalec toka na odjemalski komponenti ob izvajanju klicev na ciljno komponento preseže prag neuspešnosti za odprtje kroga, se krog odpre, prekinjevalec toka pa preide v odprto stanje. V odprtem stanju se zahteve na ciljno komponento ne izvajajo. Največkrat se v odprtem stanju prekinjevalca toka uporablja vzorec hitrega neuspeha, kar pomeni takojšnji odziv na zahtevo zaradi napake. Po izteku čakalnega časa prekinjevalec toka preide v polodprto stanje, kjer ponovno poskusi s posredovanjem manjšega števila zahtev na ciljno komponento. Če zahteve dosežejo prag uspešnosti, prekinjevalec toka preide v zaprto stanje, krog je spet zaprt, zahteve pa se normalno pošiljajo. V primeru ene neuspešne izvedbe se prekinjevalec toka vrne v odprto stanje, kjer spet čaka čakalni čas pred ponovnim prehodom v polodprto stanje [35, 36]. Na sliki 3.1 lahko vidimo diagram prehajanja stanj v prekinjevalcu toka.

Prekinjevalci toka ponujajo več možnosti določanja stanja kroga. V arhitekturah mikrostoritev smo govorili o preverjanju vitalnosti mikrostoritev v

poglavju 2.3.5. Čeprav je to eden izmed mogočih enostavnejših pristopov k določanju praga uspešnosti in neuspešnosti, poda zgolj informacije o dosegljivosti storitve in najbolj osnovne podatke o njeni vitalnosti. Ne poda pa podrobnejših podatkov o sami odzivnosti storitve. Drugi način ugotavljanja je uporaba sintetičnih transakcij (ang. *synthetic transactions*). Sintetične transakcije so umetno generirane zahteve, ki se pošiljajo periodično in preverjajo pravilnost delovanja vseh funkcionalnosti ciljne storitve. Sintetične transakcije niso najbolj smiselna rešitev, saj so implementacijsko kompleksnejše in od odjemalne komponente zahtevajo globoko poznavanje vseh funkcionalnosti klicane storitve, kar pa ni nujno potrebno. Zato je veliko bolj smiselni tretji način preverjanja stanja. Ta predvideva spremljanje stanja produkcijskih zahtev, ki v vsakem primeru potujejo skozi prekinjevalec toka. Ko delež napak preseže določeno vrednost ali prag, prekinjevalec toka preklopi v odprto stanje [32]. Najpogosteje se prag neuspešnosti za odprtje kroga določi glede na delež napak in minimalnega števila izvedenih zahtev. Prag uspešnosti za zaprtje kroga se največkrat preverja prek uspešnosti ene izvedene zahteve.

Vzorec časovnika je pogosto uporabljen znotraj vzorca prekinjevalca toka. V prekinjevalec toka je časovnik umeščen kot nadzorna enota, ki omejuje čas za izvedbo zahteve na ciljno komponento. Če ob izteku časa ni odgovora na zahtevo, se napaka ob izteku časovnika prav tako šteje kot vrsta napak, ki vplivajo na doseganja praga neuspešnosti [35].

Nadomestni mehanizem

Nadomestni mehanizem je eden od vzorcev, ki se je pojavil skupaj s prekinjevalci toka. Zasledimo ga lahko tudi pod imenom tihi neuspeh (ang. *silent fail*). Njegovo uporabo največkrat zasledimo v kombinaciji s prekinjevalci toka in/ali ponovnim poskušanjem. Namesto vračanja izjem in napak se ob neuspehu omogoči izvedba nadomestnega mehanizma, ki poskuša vrniti lepši odgovor. V večini primerov je to prazen (ang. *null*) odgovor ali generiranje preprostega statičnega odgovora. V povezavi s prekinjevalci toka se lahko

nadomestni mehanizem izvaja tudi ob iztekih časovnika ali pa hitrem neuspehu, ko je krog odprt. Nadomestni mehanizem omogoča tudi naprednejši pristop, kot je klic druge odvisnosti [35, 37].

Pregrade

Pregrade so vzorec, ki omogoča izolacijo izvajanja določene akcije in ob morebitni napaki omeji širjenje napake le znotraj izoliranega območja. Koncept je prevzet iz ladij, kjer so pregrade uporabljene za ločevanje posameznih vodotesnih predelov. Pri poškodovanju ladijskega trupa pregrade poskušajo omejiti škodo s poplavljanjem le poškodovanih predelov, medtem ko ostali ostanejo nedotaknjeni.

Pregrade so v računalništvu lahko aplicirane na več nivojih, npr. na fizičnem nivoju z izvajanjem delov aplikacije v različnih fizičnih strežnikih, z uvedbo virtualizacije in vsebnikov (ang. *container*) se škoda omejuje na posamezne virtualne računalnike oz. vsebnik, na še višjem nivoju pa pregrade predstavljajo večnitno izvajanje operacij. Znotraj posameznega primerka mikrostoritve in klicih na zunanje odvisnosti pregrada največkrat predstavlja izvajanje v ločeni niti. Pri uporabi pregrad z nitnim izvajanjem se pogosto omeji tudi količina klicev na zunanjo komponento, saj je dobra praksa, da so niti za izvajanje na voljo znotraj bazena niti (ang. *thread pool*) z omejeno velikostjo. Na ta način se izognemo tako preobremenitvi zunanje komponente, kot tudi preprečimo preobremenitev odjemalske aplikacije, kjer se pregrada uporablja. Na ta način želimo preprečiti večje napake v delovanju, v najslabšem primeru prenehanje delovanja [35].

3.1.3 Samodejno okrevanje

Z vidnostjo in odpornostjo na napake je mogoče identificirati vir napake. Če so vzorci pravilno implementirani, lahko sistem sam poskrbi za čim boljšo reakcijo in delovanje ob pojavitvi napake brez večjega vpliva na končne odjemalce. Mikrostoritev, ki povzroča napake, je v večini primerov mogoče zaznati že z uporabo preprostega vzorca preverjanja vitalnosti, ki se izvaja

periodično. Če je storitev nedosegljiva, ali pa se izvor nepravilnega delovanja nahaja v izgubi povezave z zunanjo odvisnostjo, se bo to zaznalo prek pravilno implementiranega preverjanja vitalnosti. Pogosta akcija v primerih neuspešnega preverjanja vitalnosti je uporaba principa STONITH (*Shoot the Other Node in the Head*) s področja visoke razpoložljivosti in omejevanja virov. Princip v tovrstnih primerih predvideva odstranitev (ustrelitev) ranjenega primerka mikrostoritve in zamenjavo z novo oblikovanim primerkom [3, 38].

3.2 Orodja za odpornost na napake v programskem jeziku Java

V tem poglavju bomo predstavili najbolj razširjena orodja za programski jezik Java, ki implementirajo odpornost na napake. Opisali bomo specifikacijo, ki definira način uporabe in delovanja vzorcev odpornosti na napake. Orodja omogočajo uporabo več vzorcev odpornosti na napake. Pri njihovi uporabi je zelo smiselna in pogosta uporaba znotraj prekinjevalca toka, ki deluje kot osrednji vzorec. Poleg prekinjevalcev toka med vzorce odpornosti na napake spadajo še časovnik, ponovno poskušanje, hitri neuspeh, nadomestni mehanizem in pregrade. Vse predstavljene rešitve izvajajo odpornost na napake na strani odjemalca, ki izvaja klic na oddaljeno storitev.

3.2.1 Hystrix

Hystrix [6] je orodje za nadzor interakcij med porazdeljenimi storitvami prek odpornosti na napake in odpornosti na zakasnitve (ang. *latency tolerance*). Hystrix je bil razvit znotraj podjetja Netflix kot del izboljšav za prožnost sistema. Razvit je bil za velik sistem, kjer je bilo z rešitvijo na dnevni bazi izvedenih ogromno število klicev. Kasneje je knjižnica postala na voljo javnosti v okviru odprtokodnih rešitev Netflix OSS.

Orodje Hystrix na področju prekinjevalcev toka in odpornosti na napake

velja za daleč najbolj znano rešitev [10]. Polega vzorca prekinjevalca toka orodje podpira tudi vzorce časovnika, hitrega neuspeha, nadomestnega mehanizma in pregrad. Poleg omenjenih vzorcev ponuja nabor dodatnih funkcionalnosti, kot sta predpomnenje zahtev (ang. *request caching*) in združevanje zahtev (ang. *request collapsing*). Predpomnenje zahtev omogoča vračanje predpomnjenih rezultatov na podlagi definiranega predpomnilniškega ključa (ang. *cache key*). Združevanje zahtev ponuja možnost združevanja prejetih zahtev in izvajanja skupne poizvedbe na zunanjo odvisnost. Z uporabo združevanja zahtev se veliko časa pridobi predvsem pri omrežnem prenosu zahtev na zunanjo odvisnost in z zmanjšanjem obremenitve na omrežje [37].

Hystrix izvaja klice na zunanje storitve prek ukazov (ang. *command*). Ukaz je predstavljen z objektom, znotraj katerega zavijemo (ang. *wrap*) klic na zunanjo odvisnost, in predstavlja osnovni gradnik orodja. Vsak ukaz pripada ukazni skupini (ang. *command group*). Ukazne skupine so pomembne predvsem za logično združevanje ukazov, kar je uporabno pri poročanju metrik, obveščanju, nadziranju ipd. Privzeto so ukazne skupine uporabljene za izbiro bazena niti, ki mu ukaz pripada. Ukazu je mogoče določiti ločen ključ za izbiro bazena niti. Hystrix kot merilo za odpiranje kroga uporablja minimalno število izvedb (privzeto 20) ter delež neuspešno izvedenih zahtev (privzeto 50 %). Ob izpolnitvi obeh pogojev se krog odpre, vse izvedbe pa so samodejno predane vzorcu hitrega neuspeha in morebitni izvedbi nadomestnega mehanizma. Orodje ob odprtju kroga ne izvaja ukazov do izteka čakalnega časa (privzeto 5000 ms). Ob izteku časa krog preide v polodprto stanje. V tem stanju se ukaz enkrat izvede. Če je izvedba uspešna, se krog dokončno zapre, v nasprotnem primeru pa se ukaz vrne v odprto stanje [39].

Privzeto se ukazi v orodju Hystrix izvajajo v nitnem načinu. Bazen niti je tipičen primer uporabe vzorca pregrad. Z bazenom niti Hystrix omejuje ustvarjanje niti, ki bi ob hudi obremenitve lahko porabile vse sistemske vire. Bazeni niti so lahko v orodju Hystrix zelo dinamični. Vsakemu bazenu niti pripada vrsta (ang. *queue*), ki je prav tako velikostno omejena. Če so vse kapacitete zasedene, se uporabi vzorec hitrega neuspeha, ki zahtevo takoj

zavrne. Pri tem je mogoča izvedba nadomestnega mehanizma.

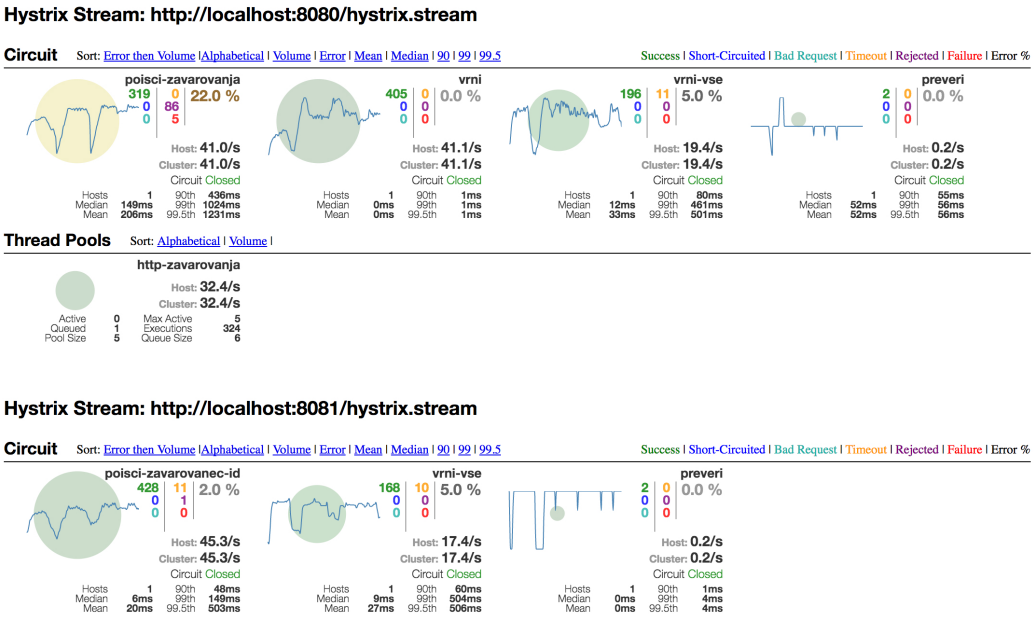
Poleg bazena niti je omogočeno tudi alternativno izvajanje ukazov v nenitnem načinu z uporabo semaforiziranega načina delovanja prek razreda `Semaphore` programskega jezika Java. S semaforji je omogočeno nadziranje števila sočasnih izvedb posameznega ukaza. Koncept semaforjev je bil v orodju Hystrix razvit predvsem za klice, ki potrebujejo čim hitrejšo izvedbo (npr. klici na podatkovno bazo) in vsebujejo zanesljive odjemalce za klice na oddaljene komponente. Klici se v semaforiziranem načinu izvedejo v isti niti, v kateri je bila zahteva predana orodju. Zaradi tega je v semaforiziranem načinu omejena uporaba časovnika. Če se klicana komponenta ne odziva, tudi sama izvedba ukaza ne more biti prekinjena [37, 39].

Ukazi se v orodju Hystrix izvajajo prek objekta, ki deduje in implementira ustrezne metode nadrazreda. Ukazi v orodju Hystrix so lahko izvedeni:

- **Sinhrono** — klasična zaporedna izvedba ukaza z uporabo metode `execute`.
- **Asinhrono** — asinhrona izvedba ukaza, kjer uporaba metode `queue` vrne objekt razreda `Future` in sproži izvedbo, končni rezultat pa dobimo s klicem metode `get`.
- **Reaktivno** — uporaba reaktivnega programiranja, kjer opazujemo rezultat prek naročanja na opazovane (ang. *observable*) objekte.

Pri implementaciji ukaza za reaktivno izvajanje moramo dedovati od drugega razreda kot v ostalih dveh primerih. Pri sinhronem in asinhronem izvajanju se kot nadrazred uporabi `HystrixCommand`, pri reaktivnem pa `HystrixObservableCommand`. Reaktivno izvajanje se razlikuje tudi v načinu izvajanja. Privzeti način delovanja je semaforiziran. Način reaktivnega delovanja namreč že samodejno poskrbi za izvedbo v ločeni niti in posledično uporaba nitnega načina delovanja ni smiselna [39].

V okviru orodja Hystrix so na voljo številne metrike. Za tekoče metrike se uporablja koncept tekočega okna (ang. *rolling window*). Okno je razdeljeno



Slika 3.2: Nadzorna plošča orodja Hystrix.

na enakomerne kose (ang. *bucket*). Privzeto je okno veliko 10 sekund in sestavljeno iz 10 enosekundnih kosov. Novi zapisi tekočih metrik se vedno generirajo v najnovejši kos, medtem ko se branja oz. izvozi metrik delajo iz celotnega okna. Pri določanju uspešnosti zahtev se prav tako uporabljajo zabeležene vrednosti iz celotnega okna.

Orodje Hystrix omogoča uporabo toka dogodkov oz. metrik. Z nadzorno ploščo orodja Hystrix lahko uvozimo tok dogodkov in v realnem času s sekundnim osveževanjem opazujemo dogajanje v ukazih. Nadzorna plošča je zelo uporabna tako v razvoju, v času finega nastavljanja (ang. *fine tuning*) konfiguracij orodja za optimalno delovanje končnega sistema, kot tudi za nadzorovanje delovanja končnega produkcijskega sistema. Primer nadzorne plošče lahko vidimo na sliki 3.2.

3.2.2 Failsafe

Orodje Failsafe [40] je lahkotna knjižnica za implementacijo odpornosti na napake. Ne vsebuje zunanjih odvisnosti in je namenjena čim bolj preprosti uporabi. Orodje nudi podporo več vzorcem za odpornost na napake. Med njimi so časovnik, ponovno poskušanje, hitri neuspeh, prekinjevalec toka in nadomestni mehanizem. V primerjavi vzorcev z orodjem Hystrix orodje Failsafe dodatno ponuja ponovno poskušanje, vendar ne nudi podpore pregradam. Uporaba orodja je enostavna, konfiguracija pa fleksibilna in razvijalcu omogoča vpeljavo lastnih načinov delovanja.

Orodje poleg sinhronega podpira tudi asinhrono delovanje. Uporaba je fleksibilna, saj z omogočenim dostopom do konteksta izvajanja lahko dinamično prilagajamo način izvajanja posameznih vzorcev. Orodje Failsafe ponuja poslušalce (ang. *listener*), prek katerih je omogočena implementacija akcij ob različnih dogodkih.

V primerjavi z orodjem Hystrix je orodje Failsafe izrazito lažja rešitev. Nivo abstrakcije pri ukazih v prekinjevalcu toka je precej nižji, tudi sama uporaba je preprostejša in hitra. Konfiguracija je preprostejša in vsebuje manj lastnosti za prilagajanje delovanja. Pragova za prehajanje med stanji v prekinjevalcu toka se določata na podlagi števila izvedb, od katerih se mora določeno število izvedb končati z napako oz. uspehom.

Za uporabo v manjšem in nekompleksnem obsegu je orodje Failsafe odlična izbira. Če je implementacija večja in je potrebna uporaba več nivojev abstrakcije, logičnega združevanja ukazov, nadzorovanje z uporabo naprednih metrik, dinamične konfiguracije ipd., pa je zagotovo veliko boljša izbira orodje Hystrix.

3.2.3 Resilience4j

Orodje Resilience4j [41] je lahkotno orodje za implementacijo vzorcev odpornosti na napake. Orodje se močno zgleduje po orodju Hystrix. Zasnovano je za uporabo v funkcijskem programiranju. Orodje je zasnovano na knjižnici

Vavr, ki predstavlja funkcijsko knjižnico za Javo 8 in poskuša zmanjšati količino kode pri uporabi javanskih pristopov k funkcijskemu programiranju ter uvaja nespremenljive (ang. *immutable*) podatkovne strukture, ki so ključne pri funkcijskem programiranju. Namesto uporabe razredov pri klicih se tako uporabljajo funkcije višjega reda, ki se izvajajo znotraj orodja oz. posameznega vzorca.

Orodje Resilience4j je zasnovano modularno. Osnovni moduli ponujajo osnovne vzorce in funkcionalnosti knjižnice, medtem ko dodatni (ang. *add-on*) moduli ponujajo razne dodatne funkcionalnosti, kot so integracija metrik, ogrodij Spring Boot ter Vert.x itd. Orodje od vzorcev odpornosti na napake ponuja časovnik, ponovno poskušanje, hitri neuspeh, prekinjevalec toka, nadomestni mehanizem in pregrade.

Med osnovne module spadata še omejevalnik tempa (ang. *rate limiter*) in predpomnenje zahtev. Prekinjevalec toka v orodju Resilience4j deluje nekoliko drugače kot v orodju Hystrix. Namesto tekočega okna uporablja t. i. krožni bitni medpomnilnik (ang. *ring bit buffer*). Namesto shranjevanja rezultatov v tekoči sekundni kos okna se vsak rezultat shrani kot preprosti bit uspešnosti v trenutno pozicijo kazalca v medpomnilniku (0 — uspešna izvedba, 1 — neuspešna izvedba). Glavna prednost pristopa je, da se rezultati izvedb lahko hranijo dalj časa in niso opuščeni ob premiku tekočega okna. Prekinjevalec toka deluje nekoliko drugače tudi pri polodprtem krogu. Namesto le ene izvedbe, na podlagi katere se odloča o prehodu v zaprto ali odprto stanje, se tudi v tem stanju uporablja ločeni krožni bitni medpomnilnik. Prekinjevalec toka se o prehodih med stanji odloča na podlagi doseganju praga za število izvedb in praga za delež napak oz. uspeha pri izvedbah [42].

Resilience4j omogoča uporabo pregrad. Te so prav tako podobne konceptu, ki je postavljen v orodju Hystrix. Orodje Resilience4j uporablja zgolj semaforiziran pristop pri zagotavljanju omejevanja sočasnosti izvajanja. Poleg tega je na voljo tudi možnost omejitve maksimalnega čakalnega časa posamezne zahteve, ko se ta želi izvesti v pregradi, ki je zasedena.

3.2.4 Specifikacija Microprofile Fault Tolerance

Specifikacija Microprofile poleg splošne specifikacije za razvoj mikrostoritev v programskem jeziku Java vsebuje ter razvija tudi specifikacije za druge projekte. Mednje spada specifikacija za uporabo vzorcev odpornosti na napake MicroProfile Fault Tolerance. V času nastajanje te naloge je bila izdana prva končna verzija specifikacije v1.0 [43]. Specifikacija obsežneje določa delovanje, funkcionalnosti in način uporabe vzorcev odpornosti na napake v arhitekturah mikrostoritev.

Uporaba vzorcev se razlikuje od preostalih predstavljenih orodij. Ker je specifikacija Microprofile namenjena uporabi standardiziranih komponent Java EE, se ta pristop uporabi tudi v tem primeru. Vzorci odpornosti na napake so zato predvideni za uporabo v javanskih zrnih CDI. Zanje so specificirane anotacije, ki definirajo način delovanja metod. Vse predpisane anotacije se lahko uporabijo na nivoju razreda ali metode. Če se anotacija uporabi na razredu, se njena uporaba prenese na vse metode tega razreda. Vse anotacije so definirane kot vezave za prestreznike (ang. *interceptor binding*), kar pomeni, da jih lahko asociiramo s prestrezniki, ki prestrežejo njihovo izvedbo v ciljnih zrnih [44]. To je standardni pristop uporabe anotacij v standardu Java EE. Specifikacija Microprofile Fault Tolerance definira naslednje anotacije (in vzorce):

- `@Asynchronous` — definira izvajanje v ločeni niti. Metoda mora vračati objekt razreda `Future`.

- `@Timeout` — definira uporabo vzorca časovnika. Pripadnika (ang. *member*) anotacije določata vrednost in enoto za časovno omejitev izvedbe metode.

- `@Retry` — definira uporabo vzorca ponovnega poskušanja. Pripadniki anotacije določajo maksimalno število ponovnih poskusov, vrednost in časovno enoto zakasnitve pred ponovnim poskušanjem, vrednost ter časovno enoto za dodajanje naključnega trepetanja, vrednost ter enoto

za maksimalno časovno omejitev vseh izvajanj, polje podrazredov razreda `Throwable`, pri katerih se izvaja ponovno poskušanje, in polje podrazredov razreda `Throwable`, pri katerih se ponovno izvajanje prekine.

- `@Fallback` — definira uporabo nadomestnega mehanizma. Pri nadomestnem mehanizmu sta mogoča dva pristopa:
  - Definicija razreda, ki implementira metodo `handle` vmesnika `FallbackHandler`. Pri tem mora biti razred prav tako zrno CDI, metoda `handle` pa mora obvezno vračati enak tip kot metoda, znotraj katere se je sprožil nadomestni mehanizem.
  - Definicija imena metode znotraj ciljnega razreda, ki se izvede ob nadomestnem mehanizmu. Metoda mora vračati enak tip in sprejeti enake argumente kot metoda, znotraj katere se je sprožil nadomestni mehanizem.
- `@CircuitBreaker` — definira uporabo vzorca prekinjevalca toka. Pripadniki anotacije določajo vrednost ter enoto čakalnega časa ob odprtju kroga, minimalno število zahtev, minimalni delež napak za doseganje praga za odprtje kroga, število uspešno izvedenih zahtev za doseganje praga za zaprtje kroga in polje podrazredov razreda `Throwable`, ki definirajo izvedbo kot neuspešno.
- `@Bulkhead` — definira uporabo vzorca pregrad. Pripadniki anotacije določajo maksimalno število vzporednih izvajanj in dolžino čakalne vrste. Mogoča sta dva pristopa k uporabi pregrad:
  - Bazen niti — način se aktivira, ko se poleg anotacije `@Bulkhead` uporabi `@Asynchronous`.
  - Semafor — uporaba semaforiziranega nadzora števila izvajanj. Uporaba čakalne vrste v tem pristopu ni mogoča.

Poleg definiranih anotacij je določena tudi integracija s konfiguracijo specifikacije MicroProfile. Vsak od opisanih vzorcev (z izjemo nadomestnega mehanizma) po specifikaciji lahko deluje samostojno ali v kombinaciji s preostalimi.

Pri vzorcu prekinjevalca toka je smiselno posebej omeniti logiko za prehajanje stanj. Krog se odpre, ko je dosežen prag minimalnega deleža napak ob izvedbah. Pri tem se upošteva izvedbe znotraj tekočega okna, ki mora vsebovati minimalno število izvedenih zahtev. V polodprtem stanju specifikacija MicroProfile omogoča več izvedb. Če je le ena neuspešna, preide prekinjevalec toka nazaj v odprto stanje. Ko izvedbe dosežejo definirano število uspešno izvedenih zahtev, prekinjevalec toka preide v zaprto stanje.

3.2.5 Povzetek orodij in specifikacije

S pregledom orodij za implementacijo odpornosti na napake smo se spoznali z usmeritvami in različnimi pristopi k uporabi prekinjevalca toka in preostalih vzorcev odpornosti na napake. Orodje Hystrix je dominantno na tem področju. Hkrati je vodilo in zgled za razvoj večine preostalih knjižnic. V implementaciji rešitve za odpornost na napake za mikrostoritve v Javi smo uporabili orodje Hystrix za nadzorovanje izvajanja oddaljenih klicev zaradi uveljavljenosti, razširjenosti, fleksibilnosti pri konfiguraciji in stabilnosti orodja. Pri tem smo uporabo orodja združili z uveljavljenimi pristopi v standardu Java EE prek specifikacije MicroProfile Fault Tolerance. Zasnovo in implementacijo rešitve bomo opisali v naslednjem poglavju.

Poglavje 4

Rešitev za doseganje odpornosti na napake v Javi

V tem poglavju bomo predstavili rešitev za doseganje odpornosti na napake v Javi, ki smo jo implementirali kot razširitev za ogrodje mikrostoritev KumuluzEE. Pri tem smo uporabili dobre pristope in orodja, ki smo jih spoznali ob pregledu že obstoječih rešitev na področju odpornosti na napake.

V poglavju 4.1 bomo predstavili zasnovo in implementacijo rešitve. V poglavju 4.2 si bomo pogledali umestitev orodja Hystrix, ki predstavlja glavnega izvajalca vzorcev odpornosti na napake v rešitvi. Rešitev smo integrirali s konfiguracijo, ki je na voljo prek ogrodja KumuluzEE, njegovo integracijo in način delovanja pa bomo predstavili v poglavju 4.3. Za veliko število podatkov, ki so na voljo v vzorcih odpornosti na napake, smo zasnovali ter implementirali metrike in jih bomo spoznali v poglavju 4.4. Na koncu bomo v poglavju 4.5 predstavili uporabo celotne rešitve na primeru.

4.1 Zasnova in implementacija rešitve

Rešitev je zasnovana kot razširitev KumuluzEE Fault Tolerance za ogrodje KumuluzEE. Cilj zasnovane rešitve je enostavna uporaba prekinjevalca toka in ostalih vzorcev odpornosti na napake prek dobro uveljavljenih orodij za

podporo vzorcem odpornosti na napake. Rešitev smo načrtovali kot modularno in je zasnovana na način, ki omogoča enostavno dodajanje podpore za uporabo različnih orodij za odpornosti na napake. V začetku smo v rešitvi KumuluzEE Fault Tolerance podprli orodje Hystrix, ki velja za najboljše in najbolj pogosto uporabljeno orodje na tem področju. Rešitev omogoča napredno nastavljanje in definiranje delovanja prek konfiguracije, osnovno uporabo vzorcev odpornosti na napake pa smo definirali z uporabo anotacij specifikacije MicroProfile Fault Tolerance.

Vzorce odpornosti na napake smo zasnovali kot anotacije, ki so skladne s specifikacijo MicroProfile Fault Tolerance. Anotacije definirajo uporabo vzorcev prekinjevalca toka (`@CircuitBreaker`), ponovno poskušanje (`@Retry`), časovnik (`@Timeout`), nadomestni mehanizem (`@Fallback`) in pregrade (`@Bulkhead`). Anotacija `@Asynchronous` določa izvajanje znotraj vzorca pregrade v ločeni niti. Podrobneje smo uporabe anotacij opisali v opisu specifikacije MicroProfile Fault Tolerance v poglavju 3.2.4. Z implementacijo prestreznika za odpornost na napake smo definirali prestrezanje klicev metod, ki so anotirane z anotacijo `@CircuitBreaker`. Omenjena anotacija je lahko uporabljena tudi na razredu. V tem primeru se prestrežejo vsi klici metod znotraj anotiranega razreda.

Rešitev za doseganje odpornosti na napake je zasnovana modularno. Skupni modul `common` vsebuje skupno kodo in jo uporabljajo ter kot odvisnost vključujejo vsi preostali moduli. V skupnem modulu se nahajajo anotacije, prestreznik, definicije vmesnikov, modelov, naštevalnih tipov (ang. *enum*), izjem, razred statičnih metod za pomoč ter glavni razred za delegiranje in upravljanje z odpornostjo na napake. Preostali moduli so izvajalni moduli in vsebujejo konkretne implementacije posameznih vzorcev prek različnih orodij za odpornosti na napake. Vsebujejo tudi pripadajoče odvisnosti za uporabo orodij. V primeru uporabe orodja Hystrix se izvajalni modul imenuje `hystrix`. Njegova zasnova in implementacija je opisana v poglavju 4.2.

Glavni razred skupnega modula je imenovan `FaultToleranceUtilImpl` ter implementira vmesnik `FaultToleranceUtil` in je zrno CDI. Ko prestre-

znik prestreže klic metode, ki je anotirana za uporabo odpornosti na napake, z vstavljanjem odvisnosti pridobi objekt glavnega razreda in mu posreduje kontekst klica (ang. *invocation context*), ki vsebuje podatke o klicani metodi, parametrih, objektu itd. Dodatno se kot parameter poda tudi kontekst zahteve (ang. *request context*), s katerim smo omogočili pravilno delovanje vstavljanja odvisnosti CDI. Podrobnejši razlogi za posredovanje parametra konteksta zahteve so razloženi v poglavju 4.2.

Ko glavni razred prejme klic za izvajanje, najprej ustvari in pridobi metapodatke o izvajanju ter jih shrani v objekt razreda `ExecutionMetadata`. Najbolj pomembna podatka sta ciljni razred (ang. *target class*) in ciljna metoda (ang. *target method*), prek katerih pridobimo večino metapodatkov. Sem spadajo predvsem anotacije metode oz. razreda, ki določajo uporabo vzorcev odpornosti na napake. V tem koraku je zelo pomembno ugotavljanje ustreznosti morebitnih dodeljenih metod za izvajanje nadomestnega mehanizma. Odvisno od načina uporabe anotacije `@Fallback`, je treba preveriti ustreznost vračanega tipa metode in vhodnih parametrov funkcij. Tudi pri uporabi anotacije `@Asynchronous` je treba preveriti ustreznost vračanega tipa ciljne metode. Podrobnosti glede zahtevanih tipov parametrov in vračanega tipa so opisane v specifikaciji MicroProfile Fault Tolerance v poglavju 3.2.4.

Pomemben del glavnega razreda je določitev ključa ukaza in ključa skupine. Tovrstno označevanje smo povzeli iz orodja Hystrix, ker omogoča kakovostno segmentacijo in označevanje različnih izvajanj oddaljenih klicev. Privzeto se za ključ ukaza uporablja ime metode, za ključ skupine pa ime razreda. Po ključu skupine se tudi primarno združujejo ukazi pri izvedbah znotraj pregrad. V primeru uporabe anotacije `@Bulkhead` na metodi se za ključ skupine dodeli ime metode skupaj z imenom razreda. Dodatna sprememba ključev, tako ukaza kot skupine, je mogoča z uporabo anotacij `@CommandKey` in `@GroupKey`. Treba je upoštevati, da se bo v primeru uporabe anotacije za spremembo ključa na metodi upošteval le, če bo na metodo vezana tudi anotacija za pregrade.

Ko se ključa za ukaz ter skupino nastavita in se določijo vse osnovne na-

stavitve, potrebne za izvajanje, se metapodatki izvajanja znotraj glavnega razreda shranijo in se ob vsakem nadaljnjem klicu metode nastavitve ne inicializirajo znova. Po pridobljenih metapodatkih za izvajanje se metapodatki, kontekst klica in kontekst zahteve posredujejo izvajalcu odpornosti na napake. Objekt izvajalca odpornosti na napake se pridobi prek vstavljanja odvisnosti. Izvajalec mora implementirati vmesnik `FaultToleranceExecutor` in mora biti zrno CDI. Ob klicu na metodo `execute` se izvajanje dodeli izvajalcu, ki vrne končni rezultat metode.

Rešitev KumuluzEE Fault Tolerance se lahko uporabi z vključitvijo ustrezne odvisnosti v projekt. Pri tem je treba vključiti artefakt, ki pripada izvajalnemu modulu orodja odpornosti na napake, ki ga razvijalec želi uporabiti. Vzorce odpornosti na napake konkretneje razvijalec uporabi z uporabo pripadajočih anotacij na želeni ciljni metodi oz. ciljnem razredu. V primeru uporabe anotacije na razredu se uporaba anotacije aplicira na vse metode razreda. Za delovanje anotacij je potrebna uporaba na zrnih CDI, sicer se prestreznik za aktivacijo vzorcev odpornosti na napake ne bo sprožil. Za uporabo vzorcev je obvezna uporaba vzorca prekinjevalca toka z anotacijo `@CircuitBreaker`, katerega se dopolnjuje z ostalimi vzorci.

4.2 Umestitev orodja Hystrix

Omenili smo, da smo rešitev razdelili v dva modula. Prvi je skupni modul, drugi pa je izvajalni modul, ki poskrbi za ustrezno izvajanje izbranih vzorcev odpornosti na napake z uporabo orodja Hystrix. Uporaba orodja Hystrix je zelo smiselna ob dejstvu, da gre za najbolj dovršeno orodje za odpornosti na napake, kar posnemajo tudi v drugih programskih jezikih. Hystrix prav tako podpira skoraj vse vzorce, ki jih definira specifikacija MicroProfile — časovnik, hitra napaka, prekinjevalec toka, nadomestni mehanizem in pregrade. Edini mehanizem, ki v Hystrixu ni podprt, a zahtevan v specifikaciji, je ponovno poskušanje. Integracija rešitve za odpornost na napake za ogrodje KumuluzEE je vsebovala ugotavljanje uporabljenih vzorcev, pretvorbo na-

stavitev iz anotacij v konfiguracijski sistem Hystrix in primerno nadzorovanje izvajanja ter ustrezno reagiranje ob napakah.

Poleg nastavljanja posameznih vzorcev prek anotacij smo omogočili tudi konfiguracije prek konfiguracije za ogrodje KumuluzEE. Več besed o integraciji konfiguracije v rešitev za odpornost na napake bomo namenili v naslednjem poglavju.

Pretvorba nastavitev v konfiguracijski sistem Hystrix poteka v dveh nivojih. Večina konfiguracije poteka na nivoju ukaza orodja Hystrix, ki določa delovanje vzorcev odpornosti na napake in način izvajanja. Ta je lahko nitni ali semaforiziran. Drugi nivo nastavitev je potreben le v primeru uporabe nitnega izvajanja za nastavitev bazena niti. Bazen niti se vedno dodeli nastavljenemu ključu skupine, ki pripada posameznemu ukazu. Osnovni podatki o ključih za ukaz Hystrix (ključ ukaza, ključ skupine in ključ za morebitni bazen niti) se nastavijo prek graditelja nastavitev `HystrixCommand.Setter`. Pri pridobivanju ključev se vedno preveri obstoj ključev. Če ključev ni, se poleg generiranja ključev sproži še inicializacija nastavitev. Ta se izvede le ob prvi uporabi in je nekoliko dalj trajajoči postopek. Postopek traja nekaj več časa predvsem ob uporabi konfiguracijskega strežnika, ker je treba preveriti konfiguracije na zunanji storitvi. Zaradi tega smo rešitev zasnovali tako, da se po vseh nastavitvah poizveduje le ob prvem klicu ukaza, nadaljnji klici pa ne vsebujejo ponovnih konfiguracij in je zato njihova izvedba hitrejša.

Inicializacija nastavitev poteka prek objektov razredov `CommandHystrixConfigurationUtil` in `ThreadPoolHystrixConfigurationUtil`. Oba dedujeta iz abstraktnega razreda `AbstractHystrixConfigurationUtil`, kjer se nahajajo že implementirane metode, ki so skupne obema podrazredoma. Podrazreda se razlikujeta predvsem v preslikovanjih iz posamezne nastavitve v rešitvi v ustrezno nastavitev znotraj orodja Hystrix in v ustreznem posredovanju pri nekaterih nastavitvah, ki so izključujoče ali pa povezane. Tu gre predvsem za razliko v dinamičnemu določanju velikosti bazena niti, ki se nekoliko razlikuje od statičnega. Enako velja pri velikosti vrste čakajočih opravil pri bazenih niti. Kako se spreminjajo dinamične nastavitve, bomo

ugotovili v naslednjem poglavju. Vse nastavitve v orodju Hystrix se nastavljajo prek orodja Archaius[1], ki je konfiguracijsko orodje v sklopu rešitev skupine Netflix OSS. Rešitev je primarno integrirana tudi v orodje Hystrix in omogoča enostavnejšo in bolj dinamično konfiguriranje.

Za izvajanje znotraj ogrodja Hystrix uporabljamo generični razred `HystrixGenericCommand`, prek katerega se izvajajo vsi ukazi. Razred nadrazredu posreduje nastavitve ključev ukaza, sam pa prejme kontekst izvajanja, kontekst zahteve in razred metapodatkov. Prek konteksta izvajanja lahko sprožimo nadaljnje izvajanje metode, ki jo je prestregel prestreznik. Metapodatki se v tej točki potrebujejo predvsem za ugotavljanje izvajanja nadomestnega mehanizma in za njegovo ustrezno izvedbo.

Nekaj problemov smo imeli pri nitnem izvajanju. Nit se namreč izvaja ločeno od glavne niti, v kateri je bila sprejeta zahteva. V ločeni niti nismo imeli dostopa do konteksta zahteve, posledično to prinese težave pri uporabi morebitnih vstavljenih odvisnosti v zrnu CDI, ki ga je prestregel prestreznik odpornosti na napake. Težave so se pojavljale le v primerih, ko sta imela oba, tako ciljno zrno CDI kot vstavljeno zrno CDI nastavljen doseg znotraj zahteve (ang. *request scoped*). Vstavljena zrna v ločeni niti zato niso bila inicializirana. Težavo smo rešili s pridobivanjem konteksta zahteve pri prestrezanju v prestrezniku in podajanjem pridobljene odvisnosti kot parameter. Če se izvajanje izvaja v ločeni niti in kontekst zahteve ni aktiviran, ga ročno aktiviramo znotraj objekta ukaza.

Orodje Hystrix ne podpira vzorca ponovnega poskušanja. V rešitvi smo se omenjeni vzorec odločili podpreti. Delovanje vzorca smo implementirali v sobivanju z orodjem Hystrix. Za podporo vzorca smo potrebovali ločeno interno hranjenje nastavitev uporabljenih vzorcev. Ker tu nismo mogli uporabiti rešitve Archaius, smo implementirali razred `RetryConfig`, ki vsebuje konfiguracije za ponovno poskušanje. Določanje vrednosti lastnostim, ki se ne morejo spreminjati naknadno, smo omogočili le ob inicializaciji objekta. Izvajalec pred izvajanjem preveri prisotnost vzorca ponovnega poskušanja

[1] `https://github.com/Netflix/archaius`

in glede na prisotnost ustrezno aktivira vzorec. Izvajanje ponovnega poskušanja skrbi za ugotavljanje uspešnosti izvajanja in izpolnjevanje pogojev ponovnega poskušanja, vsako izvajanje pa se izvaja znotraj orodja Hystrix.

4.3 Konfiguracija odpornosti na napake

Zasnovano rešitev za odpornost na napake smo integrirali z orodjem za konfiguracijo v ogrodju KumuluzEE, imenovanim KumuluzEE Config[2]. Kot smo že spoznali, orodje omogoča napredno konfiguriranje aplikacije, podprta pa je tudi uporaba konfiguracijskega strežnika.

Integracija s konfiguracijo KumuluzEE Config deluje tako, da je večina nastavitev, ki jih je mogoče nastaviti prek anotacij, mogoče nastaviti tudi prek konfiguracij. Nastavitve, ki jih prek konfiguracije s KumuluzEE Config ni mogoče nastaviti, so ključi ukazov in ključi skupin ter vključevanje posameznih vzorcev odpornosti na napake. Ključev ukaza ni mogoče spreminjati zato, ker so, kot bomo prikazali, del konfiguracijskega ključa in predvsem identifikacijski način znotraj rešitve za določanje pripadnosti določenega ukaza in skupine. Podobno smo se odločili tudi pri dodajanju posameznih vzorcev. Te je mogoče dodati le s prisotnostjo anotacije. Opisani način delovanja določa tudi specifikacija MicroProfile Fault Tolerance.

Nastavitve se prek KumuluzEE Config torej določa prek ključev. Konfiguracije v KumuluzEE Config predstavljajo drevo konfiguracij. Posamezen ključ predstavlja pozicijo v strukturi drevesa [45]. Prvi nivo v ključih za rešitev KumuluzEE Fault Tolerance je vedno tip razširitve, in sicer `fault-tolerance`. Sledijo ključ skupine, ključ ukaza, ime vzorca ter na koncu ime lastnosti vzorca, ki ji dodeljujemo vrednost. Izjema je pri konfiguraciji pregrad. Pregrade so vezane na skupino in zato definicija konfiguracij znotraj ukaza ni smiselna. Če je konfiguracija lastnosti prek KumuluzEE Config podana, bo ta preklicala nastavitev na anotaciji.

V rešitvi smo omogočili konfiguracijo lastnosti vzorca na različnih nivojih.

[2] `https://github.com/kumuluz/kumuluzee/wiki/Configuration`

Konfiguracije, vezane na ukaze, so specifične za posamezni ukaz (ang. *command specific*). Konfiguracije, vezane na posamezno skupino, so specifične za celotno skupino (ang. *group specific*). Konfiguracije lahko podamo tudi globalno, brez definiranih ključev skupine in ukaza. Globalne nastavitve veljajo za vse skupine in ukaze. Pri tem je pomemben prioritetni vrstni red. Konfiguracije, ki pripadajo ukazu, vežejo najmočneje, sledijo jim konfiguracije skupine, na koncu pa so globalne konfiguracije. Vse pa povozijo (ang. *override*) vrednosti, določene prek anotacij.

Z uporabo konfiguracije KumuluzEE Config smo omogočili tudi konfiguriranje lastnosti, ki so specifične za orodje Hystrix in so nadgradnja funkcionalnostim, ki so mogoče prek anotacij in specifikacije MicroProfile. Specifične konfiguracije za orodje Hystrix se dodaja enako kot vse ostale in zanje veljajo enaka pravila vezave lastnosti.

Za omogočanje dinamičnosti je v arhitekturi mikrostoritev izrednega pomena možnost posodabljanja vrednosti. KumuluzEE Config omogoča opazovanje konfiguracijskega ključa. Pri tem smo dodali nekaj omejitev. Posamezna aplikacija lahko vsebuje veliko število ukazov, za vsak ukaz pa je znotraj razširitve za Hystrix na voljo več kot 25 konfiguracij, od katerih je mogoče dinamično spreminjati okoli 15 konfiguracij. Zato smo se odločili dodati konfiguracijo, ki sploh omogoči naknadno spreminjanje konfiguracij, treba pa je definirati tudi kombinacije imena vzorca in pripadajoče lastnosti (npr. `timeout.value`), za katere želimo eksplicitno vključiti opazovanje. Za vključitev opazovanja je tudi potrebno, da je nastavitev ob inicializaciji ukaza (oz. skupine v primeru pregrade) prisotna. Vedno se najprej preveri morebitne konfiguracije vezane direktno na ukaz, nato na skupino in nazadnje globalne konfiguracije. Nivo, na katerem bo najdena konfiguracija za posamezen ukaz, bo tudi opazovan. Če se torej na nek ukaz dodeli lastnost, definirana na globalni ravni, in je opazovana, se morebitna pojavitev konfiguracije na nivoju ukaza ne bo upoštevala. Nova konfiguracija bi se upoštevala šele ob ponovnem zagonu aplikacije.

Tudi tu smo morali ločeno poskrbeti za posodabljanje vrednosti pri vzorcu

ponovnega poskušanja, ki v orodju Hystrix ni podprt. Načela konfiguracije so tu popolnoma enaka, razlikuje se le interna implementacija. Uporablja se ločen nadzornik konfiguracij za ponovno poskušanje. To je potrebno zato, ker ostale konfiguracije upravljamo prek orodja Archaius, pri ponovnem poskušanju pa za konfiguracije v celoti skrbi ločena implementacija. Posodabljanje konfiguracij je tako pri ponovnem poskušanju drugačno od preostalih, zato smo se tudi odločili za strožjo ločitev kode. Konceptualno sicer pri delovanju večjih razlik med pristopoma ni.

4.4 Metrike odpornosti na napake

Prekinjevalci toka z uporabo preostalih vzorcev odpornosti na napake predstavljajo izvor velike količine podatkov o dostopnosti in odzivnosti klicanih odvisnosti. Orodje Hystrix je pri tem še posebej bogato. Vsebuje podatke o številu izvedb zahtev znotraj tekočega okna, številu napak, na podlagi katerih je mogoče izračunati število izvedenih zahtev na sekundo, ter delež napak. Poleg tega so na voljo podatki o številu različnih dogodkov, mogoči so podatki tako o številu znotraj tekočega okna kot tudi števci, ki zajemajo celoten življenjski čas aplikacije. Med pomembnejšimi so:

- **Uspešna izvedba (ang. *success*)** — število uspešnih izvedb.
- **Neuspešna izvedba (ang. *failure*)** — število neuspešnih izvedb.
- **Iztek časovnika (ang. *timeout*)** — število izvedb, ki so se končale s pretekom časovnika.
- **Kratek stik (ang. *short-circuited*)** — število izvedb, ki so bile takoj zavrnjene z vzorcem hitrega neuspeha ob odprtem krogu.
- **Zavrnitev bazena niti (ang. *thread pool rejected*)** — število izvedb, ki so bile zavrnjene zaradi polne zasedenosti bazena niti in (morebitne) čakalne vrste.

- **Semaforska zavrnitev (ang. *semaphore rejected*)** — število izvedb, ki so bile zavrnjene zaradi polne zasedenosti pri semaforski izvedbi.
- **Nepravilna zahteva (ang. *bad request*)** — število izvedb, ki so se končale z nepravilno zahtevo in ne kot neuspešna izvedba.
- **Uspešna izvedba nadomestnega mehanizma (ang. *fallback success*)** — število izvedb, ki so se končale z uspešno izvedbo nadomestnega mehanizma ob napaki v osnovni izvedbi.
- **Neuspešna izvedba nadomestnega mehanizma (ang. *fallback failure*)** — število izvedb, ki so se končale z neuspešno izvedbo nadomestnega mehanizma ob napaki v osnovni izvedbi.
- **Zavrnitev izvedbe nadomestnega mehanizma (ang. *fallback rejected*)** — število izvedb, ki so se končale z zavrnitvijo izvedbe nadomestnega mehanizma.
- **Oddaje (ang. *emit*)** — število oddanih zahtev.
- **Oddaje nadomestnega mehanizma (ang. *fallback emit*)** — število oddanih zahtev za izvedbo nadomestnega mehanizma.

Pridobimo lahko izvedbene čase posameznega ukaza. Ti časi so na voljo kot centili oz. percentili, kot so imenovani v orodju Hystrix. 100p-ti centil, kjer je p med 0 in 1, je definiran kot število, od katerega ima 100p odstotkov meritev manjšo ali enako numerično vrednost [46]. Orodje Hystrix ponuja kar nekaj izračunanih centilov. Pri implementaciji metrik smo se odločili za centile reda 50, 75, 90, 95, 99 in 99,9.

Uporaba metrik prek ogrodja KumuluzEE je mogoča prek orodja za metrike KumuluzEE Metrics. Metrike so v ogrodju KumuluzEE zasnovane na podoben način kot odpornost na napake. V osnovnem modulu `core` se nahajajo osnovna orodja za registracijo metrik v registre. Dodatno se na voljo še trije moduli za izvoz metrik v različne produkte. Podprti so izvozi v orodja

Prometheus, Graphite in Logstash. Pri metrikah se kot osnovno orodje uporablja uveljavljeno orodje Dropwizard Metrics[3].

V orodju Dropwizard Metrics register metrik predstavlja množico metrik, ki so logično združene pod istim imenom. Pod različnimi imeni se znotraj registra lahko hrani različne tipe metrik [47]. Osnovni tipi metrik orodja Dropwizard Metrics so:

- **Merilec (ang. *gauge*)** — najpreprostejši tip metrike, ki vrača vrednost.
- **Števec (ang. *counter*)** — naraščajoča vrednost, ki šteje. Štetje se začne pri 0.
- **Histrogram** — meri porazdeljenost vrednosti v toku.
- **Meter** — meri tempo, pri katerem se neka množica dogodkov odvija.
- **Štoparica (ang. *timer*)** — neke vrste histogram trajanja določenega tipa dogodka in meter, ki določa tempo odvijanja dogodka.

Rešitev za metrike znotraj odpornosti na napake smo torej zasnovali kot razširitev KumuluzEE Fault Tolerance Hystrix Metrics. Integracija z orodjem Hystrix je relativno preprosta, saj je orodje zelo dobro zasnovano in omogoča enostavno registracijo vtičnikov, med katerimi je tudi podpora za registracijo vtičnikov za izdajatelja (ang. *publisher*) metrik. Izdajatelja je treba registrirati pred prvim klicem ukaza znotraj orodja Hystrix. To je enostavno izvedljivo z metodo `init`, saj prek nje ogrodje KumuluzEE inicializira vsako registrirano razširitev. Pri inicializaciji vtičnika je treba podati ime registra. To se lahko poljubno določi prek konfiguracije KumuluzEE Config. Privzeto se za register uporabi ključ `fault-tolerance-hystrix`. Orodje KumuluzEE Metrics omogoča pridobitev objekta za želeni ključ registra, register pa se hkrati inicializira in je s tem na voljo pri izpostavitvi različnim sistemom za upravljanje metrik.

[3] `http://metrics.dropwizard.io`

Ko sta bila registrirana tako register za orodje KumuluzEE Metrics kot vtičnik v orodju Hystrix, je bilo treba le še poskrbeti za konkretne metrike. V razred, ki predstavlja vtičnik za orodje Hystrix, je treba registrirati razreda, ki predstavljata metrike za ukaz in bazen niti. Razreda razširjata razred `HystrixMetricsPublisherCommand` za ukaz oz. `HystrixMetricsPublisherThreadPool` za bazen niti. Prek metode `initialize` smo registrirali posamezne metrike v register. Vse metrike so tipa merilec, saj že samo orodje Hystrix skrbi za ustrezno štetje, razvrščanje in merjenje dogodkov. Registrirali smo štetje dogodkov, izvajalne čase, prejete zahteve, delež napak in nekatere pomembnejše nastavitve (velikost bazena niti, velikost vrste, minimalno število zahtev za prekinitev kroga, minimalni delež napak, iztek časovnika, čakalni čas v odprtem krogu ipd.).

Rešitev KumuluzEE Fault Tolerance Hystrix Metrics definira prisotnost artefakta `hystrix` iz rešitve KumuluzEE Fault Tolerance in artefakta `core` iz orodja KumuluzEE Metrics, ki sta nastavljena kot zagotovljena (ang. *provided*), kar pomeni, da se pričakuje prisotnost omenjenih artefaktov Maven ob izvajanju končne aplikacije [48]. Z drugimi besedami povedano, končna aplikacija ne more delovati brez eksplicitno vključenih ustreznih artefaktov Maven za omenjeni orodji. To pa je tudi smiselno, saj mora razvijalec v prvi vrsti uporabljati orodji za odpornost na napake in splošne metrike, da lahko izpostavlja metrike, ki so specifične za vzorce odpornosti na napake.

4.5 Primer uporabe rešitve KumuluzEE Fault Tolerance

V tem poglavju bomo na primeru, ki je implementiran z ogrodjem KumuluzEE, prikazali, kako se pri klicu na zunanjo odvisnost enostavno uporabi implementirano rešitev za doseganje odpornosti na napake. Del konfiguracij bomo določili prek konfiguracije KumuluzEE Config in dokumenta YAML. Na primeru je uporabljeno orodje Maven za upravljanje projekta. Prikazali bomo uporabo pri enostavnem vmesniku REST, ki bo prek metode GET

Odsek kode 4.1: Dodajanje odvisnosti KumuluzEE Fault Tolerance Hystrix v dokument POM.

```
<dependency>
  <groupId>com.kumuluz.ee.fault.tolerance</groupId>
  <artifactId>kumuluzee-fault-tolerance-hystrix</artifactId>
  <version>1.0.0-SNAPSHOT</version>
</dependency>
```

vračal podatke o entiteti `Zavarovanje`. Del entitete predstavlja seznam pripadajočih entitet `ZavarovanjeKritje`, ki jih pridobimo prek zunanje odvisnosti. Klic na zunanjo storitev se bo izvedel prek protokola HTTP. Nad metodo, ki izvaja klic, bomo uporabili različne vzorce odpornosti na napake.

Za delovanje rešitve KumuluzEE Fault Tolerance je treba najprej dodati odvisnost na artefakt Maven. To dosežemo prek dokumenta POM (*Project Object Model*). Dodali bomo odvisnost na artefakt, ki vsebuje izvajalca Hystrix iz rešitve KumuluzEE Fault Tolerance. V odseku kode 4.1 lahko vidimo dodano odvisnost.

Znotraj projekta smo ustvarili preprost vmesnik REST, ki poskrbi za pridobitev entitete `Zavarovanje` in pripadajočega seznama entitet `ZavarovanjeKritje`, ki se pridobi iz zunanje odvisnosti. V odseku kode 4.2 lahko opazimo, da smo vstavili zrno CDI, ki bo poskrbelo za pridobivanje ustreznih pripadajočih entitet s klicem na zunanjo odvisnost.

V odseku kode 4.3 je prikazana implementacija zrna CDI, ki uporablja vzorce odpornosti na napake. Povozili smo privzeta ključa ukaza in skupine z anotacijama `@CommandKey` ter `@GroupKey`. Na nivoju razreda, kjer smo določili ključ skupine, smo dodali tudi vzorec pregrad z anotacijo `@Bulkhead`. Na metodi, ki poskrbi za izvedbo HTTP klica, smo dodali vzorec prekinjevalca toka z anotacijo `@CircuitBreaker`. Določili smo, da v primeru napake razreda `IOException` ne želimo izvedbe nadomestnega mehanizma. Če bo prišlo do napake drugega razreda, se bo izvedel podani nadomestni mehanizem, katerega implementacija vrača prazen seznam. Dodali smo tudi vzorec časovnika z anotacijo `@Timeout`. Izvajanje metode je semaforizirano.

Odsek kode 4.2: Implementacija preprostega vmesnika REST za pridobivanje entitete `Zavarovanje`.

```
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
@Path("zavarovanja")
public class ZavarovanjeVir {

  @Inject
  private HttpZavarovanjeKritjeZrno zavarovanjeKritjaZrno;

  @GET
  @Path("{id}")
  public Response get(@PathParam("id") Integer id) {

    Zavarovanje zavarovanje = queryZavarovanje(id);
    zavarovanje.setZavarovanjeKritja(zavarovanjeKritjaZrno
      .poisciKritja(id));

    return Response.ok(zavarovanje).build();
  }
}
```

Za podrobnejše podajanje nastavitev smo uporabili konfiguracijo KumuluzEE Config. Konfiguracije smo podali prek dokumenta YAML `config.yml`, ki se mora nahajati v razredni poti projekta. V odseku kode 4.4 lahko vidimo nastavljene konfiguracije. Globalno smo za vse prekinjevalce toka nastavili čakalni čas ob odprtju kroga na 3 sekunde in prag neuspešnosti, ki zahteva 30 % delež napak za odprtje kroga. Na skupini smo pregradam omejili sočasnost na 5 vzporednih izvajanj. Na ukazu `find-kritja` smo časovnik nastavili na 1500 milisekund. Dodatno smo omogočili tudi opazovanje sprememb, in sicer za spremembo vrednosti časovnika in deleža napak pri prekinjevalcih toka.

Če želimo izpostaviti metrike, ki jih zajema orodje odpornosti na napake, je treba v dokument POM projekta dodati še artefakte Maven za metrike KumuluzEE Metrics in artefakt za metrike odpornosti na napake orodja Hystrix KumuluzEE Fault Tolerance Hystrix Metrics. V odseku kode 4.5 lahko vidimo dodane odvisnosti. Pri izvozu metrik smo uporabili artefakt za izpostavitev metrik prek servleta za orodje Prometheus.

Odsek kode 4.3: Implementacija zrna CDI za izvedbo zahteve HTTP na zunanjo odvisnost in iskanje entitet `ZavarovanjeKritje`.

```
@RequestScoped
@GroupKey("http-zavarovanje-kritje")
@Bulkhead
public class HttpZavarovanjeKritjeZrno {

  @CircuitBreaker(failOn = {IOException.class})
  @Fallback(fallbackMethod = "poisciKritjaFallback")
  @CommandKey("find-kritja")
  @Timeout
  public List<ZavarovanjeKritje> poisciKritja(Integer id) {
    // ...
    // izvedba http klica za pridobitev kritij glede na podan ID zavarovanja
    // ...
  }

  public List<ZavarovanjeKritje> poisciKritjaFallback(Integer id) {
    return new ArrayList<>();
  }
}
```

Odsek kode 4.4: Konfiguracija odpornosti na napake prek dokumenta YAML `config.yml`.

```
fault-tolerance:
  config:
    watch-enabled: true
    watch-properties: timeout.value,circuit-breaker.failure-ratio
  circuit-breaker:
    delay: 3s
    failure-ratio: 0.3
  http-zavarovanje-kritje:
    bulkhead:
      value: 5
    find-kritja:
      timeout:
        value: 1500ms
```

Odsek kode 4.5: Dodajanje metrik odpornosti na napake v dokument POM.

```
<dependency>
  <groupId>com.kumuluz.ee.metrics</groupId>
  <artifactId>kumuluzee-metrics-core</artifactId>
  <version>1.0.0-SNAPSHOT</version>
</dependency>
<dependency>
  <groupId>com.kumuluz.ee.metrics</groupId>
  <artifactId>kumuluzee-metrics-prometheus</artifactId>
  <version>1.0.0-SNAPSHOT</version>
</dependency>
<dependency>
  <groupId>com.kumuluz.ee.fault.tolerance.metrics</groupId>
  <artifactId>kumuluzee-fault-tolerance-hystrix-metrics</artifactId>
  <version>1.0.0-SNAPSHOT</version>
</dependency>
```

Poglavje 5

Skalabilnost in izvajanje v oblačnih aplikacijah

V tem poglavju bomo pregledali skalabilnost in različne možnosti skaliranja, ki obstajajo znotraj oblačnih aplikacij. Osredotočili se bomo na izvajalna okolja v arhitekturi mikrostoritev, ki zaradi lahkotnosti posameznih aplikacij stremijo k lahkotnosti tudi na področju izvajalnih okolij. Na podlagi načinov izvajanja bomo spoznali pristope k izvajanju posameznih postavitev mikrostoritev in se spoznali z orodjem, ki omogoča orkestracijo izvajalnih okolij mikrostoritev in s tem tudi skaliranje posameznih primerkov mikrostoritev.

V poglavju 5.1 bomo pogledali skalabilnost oblačnih aplikacij, ki jo bomo v poglavju 5.2 umestili v različna izvajalna okolja mikrostoritev. V poglavju 5.3 bomo govorili o orkestraciji vsebnikov v arhitekturi mikrostoritev.

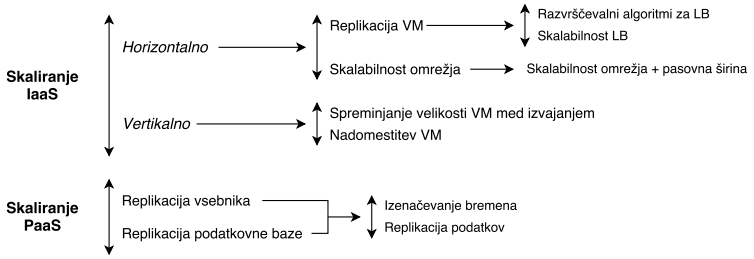
5.1 Skalabilnost v oblačnih aplikacijah

Skalabilnost je pomemben atribut omrežja, sistema ali procesa. Koncept opisuje zmožnost sistema, da se prilagodi zvišanju števila uporabnikov ali objektov, se ustrezno prilagodi povečanemu obsegu enot dela ter je dovzeten za povečanje in rast. Sistem, ki se ne skalira ustrezno, slabo vpliva na stroške izvajanja ali pa škodi kakovosti storitve [49]. Skalabilnost je izrednega

pomena v oblačnih aplikacijah in porazdeljenih sistemih mikrostoritev.

Računalništvo v oblaku v teoriji ponuja neomejene računske vire in obračunavanje storitev glede na porabo virov. Zaradi omenjenega koncepta je v oblačnih aplikacijah zelo pogosta praksa prilagajanja uporabe virov trenutnim potrebam. Z drugimi besedami, ko je obremenitev nizka, je v interesu vseh, tako ponudnika kot odjemalca oblačnih storitev, da se uporablja manj virov. Ponudnik na ta način lahko prerazporedi vire tja, kjer jih v danem trenutku bolj potrebuje, odjemalec pa na drugi strani ne povečuje stroškov po nepotrebnem. V času visoke obremenitve je treba vire povečati, da sistem lahko primerno poskrbi za vse zahteve, ne da bi pri tem trpela kakovost storitve. Storitve zato želijo samodejno skalirati svoje vire glede na obremenitve. Samodejno skaliranje tipično poteka na podlagi določenih pravil, ki ob upoštevanju različnih pogojev prilagajajo obremenjenost. Različni ponudniki in tipi oblačnih platform ter aplikacij ponujajo različne načine pogojev in pravil za skaliranje — od preprostih na podlagi porabe procesorske moči in pomnilnika do naprednih lastnih, kot so kompleksnejši pogoji in upoštevanje lastnih metrik [50].

Pristopa k skaliranju storitev sta dva — vertikalni in horizontalni. Vertikalno skaliranje povečuje računske vire npr. s povečanjem razpoložljivih virov CPU ali pomnilnika RAM strežnika. Večina operacijskih sistemov te vrste skaliranja ne podpira med samim delovanjem (ang. *on-the-fly*), zato je potreben ponoven zagon sistema za uveljavitev sprememb. Horizontalno skaliranje namesto povečanja računskih virov doda nov primerek npr. virtualni strežnik. Pri horizontalnem skaliranju je treba pravilno pristopati ne samo do ustvarjanja novih primerkov, ampak je treba poskrbeti tudi za ustrezno omrežno podporo. Pri tem je prvi v vrsti izenačevalec bremena, ki poskrbi za ustrezno razporejanje bremena med virtualnimi napravami. Poskrbeti je treba tudi za omrežno strukturo, ki ob povečanju bremena in dodajanju novih primerkov lahko ni več sposobna prenesti povečanega bremena in potrebuje dodatne povezave, povečanje pasovne širine (ang. *bandwidth*), usmerjevalnike ipd. [50, 49].



Slika 5.1: Prikaz skaliranja v oblačnih storitvah Iaas in PaaS.

Pri monolitnih aplikacijah je najpogostejši način skaliranja vertikalno skaliranje. Ob povečanju potrebe in dosega limita računskih kapacitet strežnika se strežniku poveča zmogljivost. V oblačnih aplikacijah lahko k skaliranju pristopimo tako z vertikalnim kot horizontalnim pristopom. Pristop k skaliranju se razlikuje glede na tip uporabljane oblačne storitve. Infrastruktura kot storitev IaaS (*Infrastructure as a Service*) razpolaga z virtualno strojno opremo, kot so virtualne naprave ter različna omrežna oprema. Drugačno abstrakcijo prinaša platforma kot storitev PaaS (*Platform as a Service*), kjer je vedno bolj popularen pristop s ponudbo okolja vsebnikov. Znotraj okolja PaaS so že v osnovi na voljo različna orodja in storitve, ki podpirajo aplikacijo, sestavljeno iz več komponent, in nudijo orodja, ki pomagajo pri povezovanju komponent.

IaaS in PaaS sta lahko skalirana. IaaS omogoča tako vertikalno skaliranje s spreminjanjem velikost virtualnih naprav ali kar nadomestitvijo kot tudi horizontalno skaliranje z repliciranjem virtualnih naprav, ustrezno omrežno skalabilnostjo in uporabo izenačevalca bremena. V storitvi PaaS je pogostejši pristop horizontalnega skaliranja števila vsebnikov, pri čimer orodja, ki jih ponuja PaaS, poskrbijo za ustrezno obremenitev primerkov vsebnika in se uporabnikom storitve ni treba ukvarjati z nadzorovanjem in upravljanjem ustreznosti skaliranih vsebnikov [50]. Na sliki 5.1 lahko vidimo prikaz skaliranja v oblačnih storitvah Iaas in Paas.

Oblačne aplikacije torej omogočajo več pristopov k skaliranju. Zadnji napredki v tehnologijah vsebnikov, ki ponujajo še večjo hitrost, skalabilnost in

manjše režijske stroške, so privedli do zelo velike popularnosti uporabe vsebnikov v povezavi z mikrostoritvami. Skupaj namreč še povečujejo agilnost rešitev, enostavnost delovanja, lahkotnost rešitev in izredno hitro zamenljivost [51].

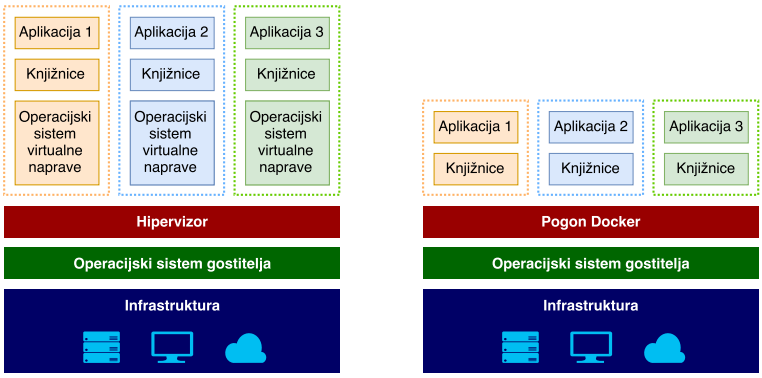
5.2 Izvajalno okolje mikrostoritev

Monolitne aplikacije se tipično izvajajo na N strežnikih, ki so lahko fizični ali virtualni. Na vsakem strežniku se lahko izvaja M primerkov aplikacije. Postavitev aplikacije ni vedno povsem enostavna, vsekakor pa je postopek veliko lažji kot pri arhitekturi mikrostoritev. Sistem mikrostoritev je sestavljen iz velikega števila manjših aplikacij. Vsaka mikrostoritev je lahko napisana v drugem programskem jeziku, predstavlja svojo postavitveno enoto, uporablja različna orodja, uporablja različne odvisnosti in je skalirana ločeno. Vsaka mikrostoritev ima lahko tudi drugačne računske zahteve — viri CPU, pomnilnik RAM in različne vhodno-izhodne (ang. *input/output*) zahteve.

Za izvajanje mikrostoritev so tipično mogoča tri okolja — fizični strežniki, virtualne naprave in vsebniki. Ker v oblačnih aplikacijah, ki se izvajajo v javnih oblakih, nimamo dostopa do samega fizičnega strežnika, ta rešitev večinoma ne pride v poštev. V javnih oblakih vse temelji na virtualizaciji. V nadaljevanju poglavja bomo zato najprej pogledali specifičnosti virtualnih naprav in vsebnikov, nato pa bomo podrobneje pregledali pristope k izvajanju mikrostoritev.

5.2.1 Virtualne naprave in vsebniki

Virtualna naprava VM (*Virtual Machine*) je emuliran računalniški sistem, ki prinaša funkcionalnosti fizičnega računalnika. Virtualne naprave poganjajo hipervizorji (ang. *hypervisor*). Hipervizor je programska oprema, ki se izvaja na fizični napravi in ponuja izolacijo virtualnih naprav ter izvaja različna jedra (ang. *kernel*) operacijskih sistemov. Virtualna naprava ob uporabi deluje in se obnaša kot navadna fizična naprava. Znotraj virtualne naprave



Slika 5.2: Virtualizacija virtualnih naprav in virtualizacij vsebnikov v pogonu Docker.

se lahko izvajajo aplikacije [52].

Napredek v jedrih operacijskega sistema Linux je privedel do virtualizacije izvajanja v vsebnikih (ang. *container based virtualization*). Vsebniki oz. njihove slike so lahkotni, samostoječi izvršljivi paketi programske opreme, ki vsebujejo vse, kar potrebujejo za svoje izvajanje — programsko kodo, izvajalnik kode (ang. *runtime*), sistemska orodja, sistemske knjižnice in konfiguracije. Pri tem je vsebnik izoliran od preostalega okolja v operacijskem sistemu. Z izvajanjem v peskovniku (ang. *sandbox*) se poskuša preprečiti konflikt z drugimi izvajajočimi programskimi rešitvami v operacijskem sistemu [53]. Primerjavo virtualizacije virtualnih naprav in vsebnikov lahko vidimo na sliki 5.2.

Vsebnik LXC (*Linux Containers*) je virtualizacijski pristop v operacijskem sistemu Linux, s katerim je mogoče izolirano izvajati več procesov znotraj istega jedra operacijskega sistema. Izolacija je mogoča zaradi naslednjih pomembnejših funkcionalnosti operacijskega sistema Linux:

- **Nadzorne skupine (ang. *control groups*)** — omogočajo omejevanje in prioritizacijo uporabe virov (CPU, pomnilnik RAM, vhodno-izhodne operacije, omrežje itd.) za skupino procesov.
- **Imenski prostor (ang. *namespace*)** — omogoča izolacijo oz. ome-

jevanje pogleda aplikacije na okolje izvajanja. Proces lahko vidi samo procese, ki so v istem imenskem prostoru. Na ta način se omejuje pogled na drevo procesov, povezave v omrežju, uporabnike in priklopljene datotečne sisteme.

- **Datotečni sistem UFS (*Union File System*)** — omogoča uporabo slojev, pri čemer je osnovni sloj jedro operacijskega sistema, novi sloji pa dodajajo posamezne komponente (npr. spletni strežnik). Vrhnji sloj je na voljo za pisanje (ang. *writable layer*).

LXC uporablja omenjene funkcionalnosti za zagotovitev izoliranega okolja izvajani aplikaciji [53, 54, 55].

Vsebniki torej prinašajo virtualizacijo znotraj operacijskega sistema. Aplikacije, ki se izvajajo v vsebnikih, si delijo operacijski sistem, sistemske knjižnice in vire, ki so na voljo operacijskemu sistemu. Oboji, tako virtualne naprave kot vsebniki, omogočajo uporabo slik (ang. *image*). Zaradi delitve operacijskega sistema so slike vsebnikov bistveno manjše, saj mora slika virtualne naprave vsebovati še celoten operacijski sistem. Poleg tega je zagon vsebnika bistveno hitrejši kot zagon virtualne naprave, saj pri vsebniku ni treba vedno znova zaganjati operacijskega sistema, ker je ta že na voljo [53].

Docker

Docker je najbolj razširjena rešitev na področju vsebnikov. Odprtokodno orodje omogoča sistematičen pristop k avtomatizaciji hitre postavitve aplikacij znotraj vsebnikov in napredno upravljanje s slikami vsebnikov, njihovim izvajanjem ter upravljanjem vsebnikov. Docker je v začetku za izvajanje vsebnikov uporabljal zgolj virtualizacijsko tehnologijo LXC. Ta je bila kasneje zamenjana z rešitvijo libcontainer[1], ki predstavlja višji nivo abstrakcije in omogoča uporabo različnih tehnologij za virtualizacijo vsebnikov, med drugim tudi tehnologijo LXC [56]. Orodje Docker virtualizacijo razširja z vmesnikom API tako na jedru kot v aplikaciji. Vsebniki Docker se ustvarijo

[1] `https://github.com/docker/libcontainer`

iz osnovne slike (ang. *base image*), ki se izbere glede na potrebe aplikacije. Pri gradnji slik z orodjem Docker se vsak ukaz oz. akcija pri gradnji pretvori v sloj (ang. *layer*), ki se zgradi nad prejšnjim slojem. Akcije se tipično izvajajo prek skriptnih datotek, imenovanih `Dockerfile`. Vsaka skripta vsebuje serijo ukazov, ki zaporedno združujejo sloje in osnovno sliko ter na koncu ustvarijo novo sliko vsebnika [53].

5.2.2 Pristopi k izvajanju mikrostoritev

Pri izvajanju mikrostoritev obstajajo naslednji osnovni pristopi k izvajanju postavitev mikrostoritev [57]:

- **Več primerkov mikrostoritev na gostitelja** — izvajanje več primerkov storitev na istem gostitelju, ki je fizična ali virtualna naprava. Predstavlja tradicionalen pristop pri postavitvi aplikacij. Njegove prednosti so enostavnejše upravljanje virov, deljenje operacijskega sistema, enostavna ter hitra zamenjava postavitve in hitri zagon storitve zaradi neobstoja dodatnih režijskih stroškov. Pristop prinaša tudi precej slabosti, med drugim neobstoj izolacije postavitev, potrebno poznavanje podrobnosti objave vsakega primerka storitve in težavno nadziranje virov, ki jih porabi posamezna postavitev.

- **Primerek mikrostoritve na gostitelja** — izvajanje enega primerka storitve na gostitelju, ki je virtualna naprava. Za vsako postavitev obstaja slika operacijskega sistema z nameščeno aplikacijo, ki se namesti na virtualno napravo. Prednosti pristopa so popolna izolacija posameznega primerka, omejevanje porabe virov za primerek in možnost uporabe zelo dovršenih oblačnih storitev (izenačevanje bremena, samodejno skaliranje). Slabosti pristopa so večji režijski stroški ob postavitvi virtualnih naprav ter s tem počasni zagon novega primerka storitve, počasna izgradnja slik postavitev, fiksna omejenost virov za posamezne velikosti virtualnih naprav pri večini oblačnih ponudnikov

in obračunavanje porabe virov po virtualnih napravah ne glede na to, ali so te zasedene ali mirujejo.

- **Primerek mikrostoritve na vsebnik** — izvajanje enega primerka storitve znotraj vsebnika. Vsebniki so sestavljeni iz enega ali več procesov, ki se izvajajo v peskovniku. Prednosti pristopa so zelo podobne kot pri pristopu s primerkom mikrostoritve na gostitelja. Razlika je predvsem v slikah. Slike vsebnikov so zelo lahkotne, njihova gradnja je hitra, zagon vsebnika je bistveno hitrejši kot pri virtualni napravi. Slabosti vsebnikov so pri zrelosti in stabilnosti tehnologije. Zrelost virtualnih naprav je na precej višjem nivoju kot pri vsebnikih. Rešitve za vsebnike v oblakih so še relativno nove (Google Container Engine, Amazon EC2 Container Service). V marsikaterem oblaku je naloga razvijalcev, da vzpostavijo infrastrukturo za vsebnike na infrastrukturi virtualnih naprav, kar zopet prinese probleme s stroški.

Navkljub omejitvam vsebnikov se ti vse bolj uveljavljajo. Zaradi enostavnosti, lahkotnosti, velike skalabilnosti, hitrega skaliranja in hitre gradnje vsebniki prevzemajo pobudo pred virtualnimi napravami. Prednost vsebnikov je tudi gostota mikrostoritev. Na posameznem gostitelju je mogoče gostiti veliko več aplikacij v primeru uporabe vsebnikov kot v primeru uporabe virtualnih naprav. Tudi rešitve na področju vsebnikov in orkestracije vsebnikov so vedno bolj dovršene in zrele. Oblačni ponudniki širijo svoj nabor ponudb na oblakih v smeri vse večje podpore storitvam za obvladovanje in izvajanje aplikacij v vsebnikih [52].

Zaradi vseh naštetih prednosti smo se tudi v tej nalogi odločili uporabiti vsebnike in rešitev Docker za testiranje arhitekturnega pristopa k skaliranju prek podatkov iz odpornosti na napake. Dodatno k temu pripomore tudi enostavnost na področju izvajanja skaliranja vsebnikov v gruči prek orkestratorja vsebnikov Kubernetes, o katerem bomo govorili v naslednjem poglavju.

5.3 Orkestracija vsebnikov v arhitekturi mikrostoritev

Uveljavljanje arhitekture mikrostoritev, ki se izvajajo v vsebnikih, je privedla do potrebe po orkestratorju vsebnikov za obvladovanje operiranja velikega števila vsebnikov. Potrebe so se pojavile tudi na področju povezovanja gostiteljev v gruče. Kubernetes je bil ena prvih rešitev na področju upravljavcev gruče. Kubernetes [58] je odprtokodna rešitev, ki jo ponuja podjetje Google skupaj z razširitvami na področju avtomatizacije postavitev, skaliranja in upravljanja aplikacij, ki se izvajajo v vsebnikih. Ena glavnih funkcionalnosti rešitve Kubernetes je ločevanje aplikacijskih vsebnikov od podrobnosti sistema, na katerem se aplikacija izvaja. Kubernetes podpira takojšnje izvajanje na nekaj večjih oblačnih ponudnikih. Pri tem se z nekaj nastavitvami določi vire oz. virtualne naprave, s katerimi razpolaga Kubernetes v gruči. Znotraj platforme so viri virtualnih naprav vidni kot surovi viri, aplikaciji oz. operaterju aplikacije pa se ni treba ukvarjati z razporejanjem virov, saj za to ustrezno poskrbi Kubernetes [53].

Kubernetes ni edina rešitev na področju operiranja vsebnikov in gručenja. Docker Swarm mode [59] je način uporabe rešitve Docker za upravljanje gruče gostiteljev Docker. Med funkcionalnostmi rešitve Docker Swarm mode je tako skaliranje, odkrivanje storitev, izpostavitev vrat zunanjemu izenačevalcu bremena, tekoče nadgradnje itd. Docker Swarm mode se dopolnjuje z rešitvijo Docker Compose [60], ki je namenjena definiranju in izvajanju aplikacij, ki vsebujejo več vsebnikov Docker. Vsebnike aplikacije se opiše v datoteki `docker-compose.yml`. Z uporabo vmesnika API upravljamo z vsemi vsebniki, pri čemer je na voljo večina funkcionalnosti, ki jih nudi vmesnik API za posamezne vsebnike, le da v primeru uporabe orodja Docker Compose ukaze lahko izvajamo na vseh definiranih vsebnikih hkrati.

Vse omenjene rešitve veljajo še za relativno mlade. Omenili smo že razliko v zrelosti med vsebniki in virtualnimi napravami. Razvoj na področju vsebnikov in njihove orkestracije je izjemno hiter. Če za pionirja komercial-

nega uspeha pri vsebnikih velja rešitev Docker, je na področju orkestracije vsebnikov ta rešitev Kuberenetes.

5.3.1 Kubernetes

Kubernetes je produkt podjetja Google. Razvili so ga na osnovi rešitev Borg in Omega, s katerima so že več kot 10 let operirali in orkestrirali vsebnike znotraj svoje infrastrukture. Kubernetes podpira postavitev gruče za vsebnike na več oblačnih platformah, med pomembnejšimi so GCE (*Google Compute Engine*), GKE (*Google Container Engine*), AWS (*Amazon Web Services*) in Openstack. Gruča za Kubernetes sestoji iz glavnega vozlišča (ang. *master node*) in delovnih vozlišč (ang. *worker node*), ki služijo dejanskemu izvajanju aplikacijskih vsebnikov. Gručo nadziramo prek vmesnika API na glavnem vozlišču, orodje ponuja tudi ukazno orodje CLI (*Command Line Interface*) za komuniciranje z vmesnikom API [55]. Kubernetes ponuja grafični uporabniški vmesnik v obliki spletne aplikacije.

Kubernetes je sestavljen iz več komponent [61]. Na glavnem vozlišču se nahajajo:

- ***kube-apiserver*** — izpostavlja vmesnik API za upravljanje gruče.
- ***etcd*** — konfiguracijski strežnik, v katerega se shranjujejo nastavitve in podatki za gručo.
- ***kube-controller-manager*** — izvaja nadzornike (ang. *controllers*), ki izvajajo niti v ozadju in skrbijo za rokovanje z nalogami v gruči. Vsak nadzornik predstavlja svoj proces. Med nadzornike spadajo nadzornik vozlišč, nadzornik replikacije, nadzornik končnih točk in nadzornik storitvenih računov ter žetonov.
- ***cloud-controller-manager*** — izvaja nadzornike, ki skrbijo za komunikacijo in integracijo z osnovnim ponudnikom oblaka, na katerem se gruča nahaja.

- ***kube-scheduler*** — skrbi za izbiro vozlišča, na katerem se bo izvajala posamezna izvajalna enota.

- **Dodatki** — dodatki niso obvezni, vseeno pa zelo priporočljivi. Mednje spadajo storitev DNS, grafični spletni vmesnik, nadziranje virov vsebnikov (prek metrik) in beleženje na nivoju gruče. Za metrike so privzeto uporabljane rešitve Heapster[2], cAdvisor[3], InfluxDB[4] in Grafana[5], medtem ko se za beleženje dnevniških zapisov tipično uporabi privzet sistem beleženja uporabljenega oblačnega ponudnika.

Glavno vozlišče torej sestavljajo komponente, ki upravljajo, zaznajo in se odzivajo na dogodke v gruči. Komponente so sicer lahko zagnane na katerem koli vozlišču gruče, vendar se tipično v ta namen uporabi eno, glavno vozlišče. Na tem vozlišču se ostali vsebniki ne izvajajo, ampak so temu namenjena delovna vozlišča [61]. Na delovnih vozliščih se izvajajo naslednje komponente:

- ***kubelet*** — primarni agent vozlišča. Spremlja objekte, ki so dodeljeni pripadajočemu vozlišču. Skrbi za priklop nosilca datotek (ang. *volume*), prenos potrebnih skrivnosti, izvajanje vsebnikov, izvajanje pregledovanja živosti (ang. *liveness probe*), poročanje o stanju izvajanih objektov in poročanje o stanju samega vozlišča.

- ***kube-proxy*** — omogoča abstrakcijo storitve z vzdrževanjem ustreznih omrežnih pravil in izvajanje posredovanja povezav na ustrezne izvajalne enote, ki pripadajo posamezni storitvi.

- **Izvajalec vsebnikov** — privzeto se uporablja rešitev Docker.

- ***supervisord***[6] — nadzornik procesov, ki skrbi za pravilno izvajanje agenta *kubelet* in orodja Docker.

[2] `https://github.com/kubernetes/heapster`
[3] `https://github.com/google/cadvisor`
[4] `http://influxdb.com`
[5] `https://grafana.com`
[6] `http://supervisord.org`

- ***fluentd***[7] — rešitev, ki zagotavlja posredovanje dnevniških zapisov za zagotavljanje beleženja na nivoju gruče.

V orkestratorju vsebnikov Kubernetes se komponente aplikacije predstavljajo prek objektov. Objekti v Kubernetesu so obstojne entitete. Kubernetes entitete uporablja za izražanje stanja v gruči. Z objekti je določeno, katere aplikacije se izvajajo v vsebnikih, kateri viri pripadajo aplikacijam in kakšna so pravila za delovanje ter obnašanje aplikacije. Objekti opisujejo želeno stanje, h kateremu bo orkestrator vsebnikov Kubernetes ves čas stremel in ga poskušal realizirati. Objekte je najlaže definirati prek dokumentov YAML. Vsak objekt ima dve lastnosti, ki sta prav tako objekta in sta zelo pomembni. Prvi objekt je `spec` in določa želeno stanje (ang. *desired state*), drugi pa je `status` in določa dejansko stanje (ang. *actual state*). Orkestrator vsebnikov poskuša dejansko stanje približati želenemu. Obvezne lastnosti objekta Kubernetes so še `apiVersion`, ki določa verzijo vmesnika API za objekt, `kind`, ki označuje vrsto objekta, ter `metadata`, ki unikatno opisuje objekt (znotraj je obvezna lastnost `name`).

Objekt v orkestratorju vsebnikov je enolično določen s podanim imenom in identifikatorjem UID (*unique identifier*), ki ga določi orodje Kubernetes. Vsak objekt pripada imenskemu prostoru. Če ga eksplicitno ne določimo, ta pripade privzetemu imenskemu prostoru. Objektom lahko določimo oznake (ang. *labels*), ki niso unikatne za posamezen objekt, temveč jih orodje Kubernetes uporablja za iskanje oz. ujemanje po oznakah. Objekti se lahko sklicujejo oz. izbirajo med sabo, za kar v večini primerov uporabimo izbirnik oznak (ang. *label selector*).

Zelo pomemben sklop so tudi anotacije objektov (ang. *annotations*). Te določajo morebitne potrebne metapodatke za objekte. Posamezne knjižnice ali orodja lahko anotacije objektov uporabijo za potrebe določanja lastnosti pri izvajanju. Anotacije določamo v kombinaciji ključ-vrednost (ang. *key-value*).

[7] `https://www.fluentd.org`

V nadaljevanju bomo pogledali pomembnejše tipe objektov v orkestratorju vsebnikov Kubernetes.

Strok

Objekt strok (ang. *pod*) je atomarni gradnik orkestratorja vsebnikov Kubernetes. Objekt strok predstavlja izvajajoči proces oz. enoto na gruči in enkapsulira enega oz. v nekaterih primerih več tesneje povezanih vsebnikov, ki imajo skupen unikatni naslov IP, skupne vire za hrambo in lastnosti, ki določajo, kako vsebnike urejati. Objekt strok predstavlja enoto postavitve. Docker se najpogosteje uporablja kot izvajalno okolje za vsebnike znotraj objektov strok. V samem orodju Kubernetes redko upravljamo direktno z objekti strok, saj so ti zasnovani kot minljivi. Stroki se glede na zahteve, ki jih vsebujejo vsebniki, dodelijo posameznemu vozlišču, kjer se izvajajo. Nad stroki se lahko izvaja horizontalno skaliranje [55, 62].

Za upravljanje s stroki Kubernetes uporablja nadzornike, ki skrbijo za pripadajoče stroke glede na potrebe aplikacije, zahteve replikacije, razpoložljive vire, skrbijo za uvedbe novih verzij in nudijo samo oskrbovalne (ang. *self healing*) zmožnosti. Med nadzornike spadajo npr. objekti postavitev (ang. *deployment*), nabor primerkov (ang. *replica set*) in nadzornik primerkov (ang. *replication controller*) [62].

V primeru uporabe nadzornikov se tipično uporabljajo predloge stroka (ang. *pod template*), ki definirajo strok, ki pripada posameznemu nadzorniku.

Nadzornik primerkov, nabor primerkov in postavitev

Nadzornik primerkov je objekt, ki s predlogo definira strok ter skrbi za izvajanje pravega števila primerkov stroka. V primeru prevelikega števila strokov nadzornik odvečne odstrani, v primeru premajhnega števila strokov pa jih ustrezno oblikuje. Nadzornik primerkov stroke, ki jih nadzira, najde z uporabo izbirnika oznak na podlagi ujemajočih oznak. Prek predloge lahko podamo stroke, ki jih nadzornik oblikuje [55, 62].

Nabor primerkov je objekt, ki velja za nadgradnjo nadzornika primerkov.

Glavna prednost je podpora naprednejšega ujemanja oznak, ki omogočajo nabor vrednosti pri določenem ključu oznake [62].

Postavitev je objekt, katerega uporaba je najbolj priporočena pri izbiri tipa objekta za nadzorovanje strokov. Omogoča uporabo deklarativnih posodobitev strokov prek objekta nabor primerkov. Postavitev ob tekočih posodobitvah oblikuje in odstranjuje objekte nabora primerkov za bolj učinkovito izvajanje posodobitev. S tem omogoča sočasno bivanje več verzij strokov in podpira povrnitev v prejšnjo verzijo v primeru težav s posodobitvijo [62].

Storitev

Stroki v Kubernetesu veljajo za umrljive enote, ki se stalno oblikujejo, uničujejo in prerazporejajo po vozliščih. Zaradi tega je izredno težko slediti njihovemu točnemu naslovu. V ta namen je vpeljan objekt storitev (ang. *service*). Storitev je abstrakcija, ki združuje skupino strokov glede na oznake in predstavlja zanesljiv dostop do skupine strokov. Kubernetes v ta namen na vsakem delovnem vozlišču uporablja komponento *kube-proxy*. Posredniški strežnik skrbi za ustrezno preslikovanje naslova storitve do končnega stroka, ki bo prejel zahtevo in jo izvedel. Storitev omogoča tudi preslikavanje vrat. Zapisi za dostop do storitev se zapišejo v strežnik DNS znotraj orodja Kubernetes, vsaka storitev ima svoj virtualni naslov IP [55, 63]. V Kubernetesu poznamo več načinov izpostavitve storitev z uporabo različnih tipov:

- ***ClusterIP*** — izpostavi storitev znotraj gruče na internem naslovu IP. Storitev je dostopna le znotraj gruče. Je privzeti tip storitve.
- ***NodePort*** — izpostavi storitev na vsakem vozlišču prek naslova IP vozlišča in statičnih vrat. Storitev tipa *ClusterIP*, kamor se bodo znotraj gruče usmerjale zahteve, se oblikuje samodejno.
- ***LoadBalancer*** — izpostavi storitev zunaj gruče z uporabo izenačevalca bremena ponudnika oblačne storitve. Storitvi tipa *ClusterIP* in *NodePort* se samodejno oblikujeta.

- ***ExternalName*** — preslika storitev na vsebino lastnosti *externalName* z vračanjem vnosa CNAME z njegovo vrednostjo (prek protokola DNS).

Samodejno horizontalno skaliranje strokov

Objekt horizontalnega samodejnega skaliranja strokov HPA (*Horizontal Pod Autoscaler*) omogoča določanje načina izvajanja samodejnega skaliranja strokov. Samodejno skaliranje je mogoče izvajati nad nadzorniki primerkov, naborom primerkov in postavitvijo. Skaliranje v Kubernetesu je trenutno v stabilni verziji mogoče le glede na obremenjenost CPU. Z beta verzijo vmesnika API je dodanih nekaj novih možnosti, kjer se omogoča skaliranje glede na aplikacijske metrike, pri čemer je treba izvesti konfiguracijske spremembe orkestratorja vsebnikov Kubernetes za vključitev agregacijske plasti, ki posrkbi za ustrezno obdelavo metrik. Pri objektu je smiselno določiti minimalno in maksimalno število primerkov strokov [55, 64].

Pri samodejnem horizontalnem skaliranju se ob uporabi stabilne verzije samodejnega skaliranja nastavi ciljna vrednost porabe vira CPU. Samodejno skaliranje se bo določalo glede na doseganje dejanske vrednosti v primerjavi s ciljno vrednostjo pri vseh udeleženih strokih. Na podlagi dejanske obremenjenosti se nato izračuna število strokov. Glede na omejitve števila strokov, podane v konfiguracijskih objektih, se nastavi novo število primerkov strokov, kateremu bo nadzornik primerkov skušal ustreči [64, 65].

Poglavje 6

Izboljšan model skaliranja prek metrik odpornosti na napake in validacija

V tem poglavju bomo analizirali in naslovili možnosti za skaliranje oblačnih aplikacij na podlagi metrik, zbranih iz vzorcev odpornosti na napake. Zasnovali bomo naprednejšo rešitev, ki bo z uporabo že omenjenih metrik omogočala bolj natančno zaznavanje dejanskega stanja aplikacije in s tem izboljšala elastičnost skaliranja. Zasnovano rešitev bomo ovrednotili in primerjali z uveljavljenim načinom horizontalnega skaliranja v orodju Kubernetes prek objekta horizontalnega skaliranja strokov.

V poglavju 6.1 bomo opisali zasnovo in delovanje rešitve za samodejno skaliranje primerkov mikrostoritev prek metrik odpornosti na napake. Definirali bomo umestitev vzorcev odpornosti na napake v arhitekturo oblačne aplikacije. Opisali bomo različne načine uporabe metrik, ki jih generirajo vzorci ter njihovo upoštevanje pri določanju skaliranja. Predstavili bomo testni primer, na katerem smo rešitev samodejnega skaliranja testirali in primerjali. V poglavju 6.2 bomo predstavili rezultate in diskusijo o uspešnosti zasnovane rešitve skaliranja prek metrik odpornosti na napake v primerjavi z objektom samodejnega horizontalnega skaliranja strokov.

6.1 Samodejno horizontalno skaliranje prek metrik odpornosti na napake

Z umestitvijo vzorcev odpornosti na napake v arhitekturo oblačne aplikacije želimo doseči dvoje:

1. Odpornost na napake — zgraditi želimo stabilno arhitekturo, ki se je sposobna pravilno in hitro odzivati na različne napake, ki se dogajajo v porazdeljenih sistemih zaradi načina komunikacije med komponentami. Pri tem želimo uporabiti vzorce odpornosti na napake, pri čemer je vzorec prekinjevalca toka osrednji vzorec.

2. Prožnost in skalabilnost — z metrikami, ki se pridobijo med uporabo vzorcev odpornosti na napake, želimo dobiti fleksibilno in skalabilno arhiktekturo aplikacije s prilagajanjem števila primerkov posamezne mikrostoritve.

Mikrostoritve se vedno pogosteje izvajajo znotraj vsebnikov, pri večjih aplikacijah pa je takih vsebnikov lahko zelo veliko. Za upravljanje velikega števila vsebnikov je smiselna uporaba orkestratorja vsebnikov. Zasnovana arhitektura predvideva uporabo rešitve Kubernetes za orkestriranje izvajanja posameznih mikrostoritev in njihovih primerkov. Za izvajalno okolje vsebnikov smo uporabili orodje Docker. Orkestrator Kubernetes nam poleg nadziranja in upravljanja z mikrostoritvami, ki so tipično zastopane kot eden izmed nadzornikov, omogoča tudi oblikovanje, odstranjevanje in enostavno ustrezno orkestriranje več primerkov posamezne mikrostoritve v obliki strokov. Strok torej predstavlja eno izvajalno enoto oziroma primerek mikrostoritve.

Arhitekturna rešitev predvideva umestitev vzorcev odpornosti na napake kot centralnega elementa v arhitekturo oblačne aplikacije, na podlagi katerega bomo dosegali odpornost, prožnost, elastičnost in skalabilnost. To pomeni uporabo vzorcev na vsakem mestu znotraj mikrostoritev, kjer se izvajajo klici na zunanje odvisnosti izven vsebnika izvajanja. Uporaba konkretnih vzorcev odpornosti na napake se lahko razlikuje glede na vsebinski

pomen klica na zunanjo odvisnost. Pri tem je v vseh primerih smiselna uporaba vzorca prekinjevalca toka kot glavnega vzorca, ki se ga primerno obogati s preostalimi vzorci časovnika, hitrega neuspeha, nadomestnega mehanizma, pregrad in ponovnega poskušanja. Pri vseh mikrostoritvah, ki uporabljajo vzorce odpornosti na napake, je treba izpostaviti metrike. Zasnovana rešitev omenjene metrike uporablja kot glavni vir podatkov o obremenjenosti, na podlagi katerih izvaja samodejno skaliranje primerkov mikrostoritev. Skaliranje se izvaja prek orkestratorja vsebnikov Kubernetes. Vsaka mikrostoritev je v orkestratorju predstavljena kot eden izmed tipov nadzornikov v orodju Kubernetes, pripadajoči stroki pa predstavljajo primerke mikrostoritve. Postopek izvajanja skaliranja bomo podrobneje spoznali v naslednjih poglavjih.

6.1.1 Uporaba metrik odpornosti na napake v orkestratorju vsebnikov Kubernetes

V poglavju 5.3.1 smo opisali samodejno horizontalno skaliranje strokov (HPA) kot objekt, prek katerega je v orkestratorju vsebnikov Kubernetes mogoče samodejno skalirati objekte strok. Prvotno smo načrtovali uporabo omenjenega objekta, ki bi omogočal izvajanje skaliranja prek lastnih metrik. Tu smo naleteli na težave. Objekt HPA v Kubernetesu trenutno v stabilni verziji vmesnika API omogoča skaliranje le na podlagi porabe CPU. V verziji v1.8 je v agregacijski plasti na voljo vmesnik API, ki je v verziji beta in omogoča skaliranje tudi na podlagi obremenjenosti pomnilnika RAM in lastnih metrik. V času načrtovanja rešitve samodejnega skaliranja mikrostoritev prek metrik vzorcev odpornosti na napake agregacijska plast orodja Kubernetes v fazi agregacije metrik ni zadoščala našim potrebam. Težave so se pojavile predvsem v postopku preslikavanja metrik na ustrezne ciljne objekte postavitve orodja Kubernetes, nad katerimi se izvaja samodejno skaliranje pripadajočih strokov.

Omenjeno težavo s preslikavanjem metrik lahko prikažemo na primeru preproste oblačne aplikacije s tremi mikrostoritvami A, B in C. Aplikacije so v orkestrator vsebnikov Kubernetes postavljene kot objekt postavitve, ki po-

skrbi za generiranje ustreznih strokov. Ob medsebojnih klicih mikrostoritve uporabljajo odpornost na napake in izpostavljajo pripadajoče metrike. Ugotovimo lahko, da ob pridobitvi metrik iz mikrostoritve A, ki znotraj ukaza odpornosti na napake kliče zunanjo odvisnost aplikacijo B, ne moremo razbrati, kateri je končni objekt postavitve, ki mu je treba metrike pripisati. Ključ ukaza vzorcev odpornosti na napake je namreč lahko popolnoma poljuben, prav tako ključ skupine. Mikrostoritev A lahko kliče zunanjo odvisnost B prek več ukazov. Če vključimo še mikrostoritev C, ki lahko kliče mikrostoritev B pod popolnoma drugačnimi ključi ukazov, lahko ugotovimo, da je problem preslikave ključev ukazov na ustrezne objekte postavitve težko rešljiv prek uporabe objekta HPA in trenutne verzije agregacijske plasti v orkestratorju vsebnikov Kubernetes.

Za ustrezno razreševanje preslikave metrik, ki se nanašajo na ciljni objekt postavitve v orkestratorju vsebnikov Kubernetes, je treba poiskati vse objekte strok, ki imajo med svojimi odvisnostmi ciljni objekt postavitve. Po identifikaciji strokov, ki vsebujejo metrike o ciljnem objektu postavitve, je treba identificirati še vse ukaze znotraj posameznih objektov strok, ki izvajajo zahteve na ciljni objekt postavitve. Šele ko uspemo določiti pripadajočo množico ukazov za vsak objekt postavitve, lahko začnemo z agregacijo in obdelavo metrik in odločanjem o skaliranju.

Zaradi omenjenih razlogov smo se odločili zasnovati lastno komponento za orkestrator Kuberenetes, ki zna pravilno preslikati, agregirati in uporabiti metrike ukazov za ustrezne objekte postavitve v orkestratorju vsebnikov Kubernetes. Zasnovo in implementacijo komponente bomo spoznali v naslednjem poglavju.

6.1.2 Zasnova in implementacija komponente samodejnega horizontalnega skaliranja

Objekte orkestratorja vsebnikov Kubernetes je mogoče skalirati prek vmesnika API. Da bi se izognili zamudni implementaciji vseh specifičnih klicev na vmesnik API, procesiranja odgovorov in ročnega generiranja zahtev,

smo za izvajanje ukazov na vmesnik API uporabili odprtokodni odjemalec platforme Fabric8[1] za programski jezik Java. Odjemalca smo uporabili za izdajanje ukazov za skaliranje in pridobivanje podatkov o objektih v gruči Kubernetes. Več o pridobivanju podatkov o objektih v gruči Kubernetes bomo spregovorili v naslednjem poglavju.

Zelo pomemben del zasnove je bila umestitev komponente za samodejno skaliranje. Na spletu je mogoče zaslediti primere uporabe več poenostavljenih pristopov, ki skaliranje izvajajo prek skript na napravah, ki niso del gruče orodja Kubernetes, temveč nadzor izvajajo od zunaj. Izvajanje skaliranja zunaj gruče ni smiselno. Problemi so poleg konceptualne neustreznosti tudi v pridobivanju metrik iz posameznih strokov. V gručah Kubernetes so stroki navzven nedosegljivi. Komponento za izvajanje samodejnega skaliranja smo zato umestili znotraj gruče Kubernetes in se izvaja v svojem stroku. Odločitev je smiselna tudi ob dejstvu, da se veliko komponent, ki podpirajo osnovno delovanje gruče Kubernetes in ne izvajajo jedrnih operacij na vozlišču, prav tako izvaja v vsebnikih kot stroki.

Zaradi lažjega odpravljanja napak, sledenja tako dnevniškim zapisom kot tudi metrikam ter sledenja odločitvam storitve samodejnega skaliranja, smo komponento zasnovali na ogrodju KumuluzEE. Na ta način smo lahko implementirali vmesnik REST, prek katerega so na voljo podatki o dogodkih, vključenih objektih in obravnavanih metrikah vzorcev odpornosti na napake. Dodatno smo vključili razširitvi KumuluzEE Config za konfiguracijo in KumuluzEE Metrics za oddajanje metrik, prek katerih smo nadzorovali delovanje komponente samodejnega skaliranja.

Pridobivanje podatkov o objektih orkestratorja vsebnikov Kubernetes

Znotraj komponente za samodejno skaliranje je najprej treba poskrbeti za pridobivanje podatkov o objektih v gruči Kubernetes. Pri tem so pomembni vsi stroki v gruči, ki ponujajo podatke, koristne samodejnemu skaliranju. Na

[1] `https://github.com/fabric8io/kubernetes-client`

drugi strani so pomembni objekti, ki vsebujejo informacije o načinu skaliranja. V ta namen je treba identificirati objekte, ki se lahko skalirajo, in stroke, ki vsebujejo metrike vzorcev odpornosti na napake. Za identifikacijo so torej potrebni metapodatki o objektih v orodju Kubernetes. Lastnosti objektov v orkestratorju vsebnikov Kubernetes vsebujejo objekt metapodatkov, ki smo ga opisali v poglavju 5.3.1.

Objekte, ki jih storitev samodejnega skaliranja lahko skalira, smo definirali kot skalirane objekte. Pri skaliranih objektih so podprti objekti tipa nadzornik primerkov, nabor primerkov in postavitev. Postavitve so bili primarni način definiranja mikrostoritev in pripadajočih strokov v tej nalogi. Na skaliranih objektih smo definirali naslednje metapodatkovne anotacije:

- ***ft-autoscale.io/autoscale*** — z vrednostjo `true` omogoča samodejno skaliranje pripadajočih strokov, definiranih prek predloge stroka.
- ***ft-autoscale.io/replicas-min*** — definira minimalno število primerkov stroka, podanega v predlogi stroka objekta.
- ***ft-autoscale.io/replicas-max*** — definira maksimalno število primerkov stroka, podanega v predlogi stroka objekta.
- ***ft-autoscale.io/target-value*** — definira ciljno povprečno vrednost primarne metrike.
- ***ft-autoscale.io/secondary-keys*** — definira polje metrik ukaza (ločene z vejico), ki bodo upoštevane kot sekundarne kontrolne metrike.
- ***ft-autoscale.io/secondary-target-values*** — definira polje ciljnih vrednosti sekundarnih kontrolnih metrik (ločene z vejico).

Definirali smo tudi metapodatke na strokih. Tu je treba določiti preslikave posameznih ukazov stroka na ciljno postavitev, ki je skaliran objekt. Ker stroki izpostavljajo metrike, mora komponenta vedeti, kako dostopati do njih. Na nivoju strokov smo definirali naslednje metapodatkovne anotacije:

- ***ft-autoscale.io/scrape*** — z vrednostjo `true` omogoča pridobivanje metrik iz stroka.
- ***ft-autoscale.io/port*** — definira vrata za dostop do metrik.
- ***ft-autoscale.io/path*** — definira pot za dostop do metrik.
- ***ft-autoscale.io/metrics-registry*** — definira register, ki vsebuje metrike vzorcev odpornosti na napake.
- ***ft-autoscale.io/command.[ključ-skupine].[ključ-ukaza]*** — definira preslikavanje ukaza na skaliran objekt. Pripadajoča vrednost definira ime objekta, na katerega se ukaz nanaša. Vrednost mora biti podana v poizvedovalnem načinu, ki se uporablja prek orodja CLI za orkestrator vsebnikov Kubernetes. To pomeni, da je treba podati tip objekta in ime objekta v notaciji `[tip-objekta]/[ime-objekta]`. Anotacij za preslikavo ukazov na skalirane objekte je lahko več.

Komponento samodejnega skaliranja smo konfigurirali tako, da periodično posodablja podatke o aktivnih skaliranih objektih in strokih. Podatki o strokih se osvežujejo na 30 sekund, podatki o skaliranih objektih pa na 2 minuti. Skozi testiranje smo ugotovili, da je omenjena konfiguracija najbolj smiselna. Stroki so večinoma potrebovali vsaj 30 sekund za inicializacijo in zagon, kar je pomenilo zadosten čas za zaznavo stroka še pred inicializacijo aplikacije znotraj vsebnika, nastavitev pa je zadostovala tudi za dovolj hitro odstranjevanje neaktivnih strokov. Skalirani objekti so stabilnejše strukture in se ne posodabljajo pogosto, zato tudi ni smiselno enako pogosto posodabljanje kot pri strokih. V 2-minutnem intervalu smo dosegli primeren kompromis med dovolj hitro posodobitvijo objekta v komponenti samodejnega skaliranja in zmanjšanjem obremenitve komponente s prepogostim posodabljanjem.

Strganje metrik in samodejno skaliranje

Metrike se strgajo iz strokov, ki imajo v anotacijah strganje omogočeno. V anotacijah stroka se nahajajo podatki o vratih in poti, na kateri so metrike

dostopne. Strganje metrik poteka zaporedno s prehodom čez seznam strokov. Pri pridobitvi odgovora je potrebna razčlenitev (ang. *parse*) odgovora, ki je v formatu JSON. Razčlenitev je izvedena robustno, saj se pred prvo izvedbo posameznega ukaza na stroku ne generirajo metrike za ukaz. Preprečiti je treba tudi napake ob morebitni napačni konfiguraciji ali nedosegljivosti stroka. V ta namen smo zelo robustno konfigurirali odjemalca HTTP na komponenti, ki pridobiva metrike. Nastavili smo časovnik za izvedbo na eno sekundo, čas za vzpostavitev povezave s strokom pa 200 milisekund. Ob morebitni nedosegljivosti stroka se ne izvaja ponovnega poskušanja.

Za ustrezno začasno shranjevanje metrik in njihovo agregacijo v komponenti smo zasnovali interni model. Upoštevajo se zgolj definirane metrike, podane prek anotacij. Poleg vrednosti metrike se shrani čas zajema podatkov in zaporedna identifikacijska številka izvedbe strganja metrik. S podatki o ukazih se za vsak strok ustrezno agregirajo podatki. Podatke o ukazih najprej shranimo za posamezni strok v objekt razreda `PodMetricsSet`. Zbrane podatke iz metrik ukazov je treba najprej grupirati po strokih in skaliranih objektih. Ukaze združimo za vsak strok glede na preslikan skalirani objekt. Metrike se zapišejo v objekt razreda `PodScalingObjectMetricsSet`. Po razporeditvi vseh ukazov v ustrezne skupine glede na pripadajoč skaliran objekt za posamezni strok, se ta doda v objekt razreda `ScalingObjectMetricsSet`, ki predstavlja metrike ukazov, ki pripadajo posameznemu skaliranemu objektu iz ene izvedbe strganja metrik.

Ob končani izvedbi strganja metrik se sproži postopek samodejnega skaliranja za vsak skaliran objekt. Samodejno skaliranje se določa glede na ciljno vrednost v_{tar}, ki je podana v anotacijah skaliranega objekta in dejansko trenutno vrednost v_{cur}. Ob uporabi metrike števila zahtev na sekundo smo vrednosti metrik posameznih ukazov za vsak skaliran objekt iz posameznega strganja sešteli. Dobljena vrednost je predstavljala končno grupirano vrednost gv_i i-tega strganja. Več besed bomo izbiri primarne metrike namenili v poglavju 6.1.4. Pri odločanju o skaliranju smo uporabili tekoče povprečje, ki predstavlja trenutno vrednost v_{cur}. Pri izračunu smo uporabili

podmnožico zadnjih n strganj, pri čemer s predstavlja število vseh izvedenih strganj. Uporabljene vrednosti za izračun tekočega povprečja predstavljajo tekoče okno, vrednost n pa velikost okna. Če je zbranih metrik manj od zahtevane velikosti okna, se skaliranje ne izvede. Velikost tekočega okna n je mogoče konfigurirati, v končni rešitvi pa smo uporabili vrednost 5. Na podlagi ciljne in trenutne povprečne vrednosti se določi optimalno število strokov za vsak skaliran objekt r_{opt}:

$$r_{opt} = \left\lceil \frac{v_{cur}}{v_{tar}} \right\rceil \tag{6.1}$$

$$v_{cur} = \frac{1}{n} \sum_{i=1}^{n} g v_{s-i} \tag{6.2}$$

Ob pridobitvi optimalnega števila primerkov r_{opt} se ta prilagodi glede na določeno maksimalno in minimalno število primerkov, ki se podata prek anotacij skaliranega objekta. Če se število razlikuje od trenutnega števila primerkov, se sproži še zadnji postopek preverjanja. V njem preverimo zadnji čas izvedbe skaliranja. S tem preprečujemo prepogosto izvajanje skaliranja in morebitne prehitre odločitve ob začasnem šumu, ki se ustvari pri spreminjanju števila primerkov. Pri izvajanju skaliranja navzdol smo nastavili časovno omejitev na 5 minut od zadnjega skaliranja, pri skaliranju navzgor pa smo jo omejili na 2 minuti. Večja omejitev pri skaliranju navzdol je nastavljena zaradi tega, ker razen večjega stroška ob izvajanju ni dodatne škode, če je dalj časa primerkov več. Pri skaliranju navzgor smo upoštevali, da so se vsebniki večinoma oblikovali prej kot v eni minuti, temu pa smo dodali še 1 minuto rezerve za umiritev šuma, ki nastane ob oblikovanju. Ob izpolnjevanju vseh pogojev se spremeni število primerkov na skaliranemu objektu. Orkestrator vsebnikov Kubernetes ob prejemu ukaza za izvedbo skaliranja nato izvrši ustvarjanje novih strokov prek svojega razvrščevalnika. Postopek strganja metrik in izvajanja skaliranja smo izvajali 5 sekund po končanem predhodnem strganju metrik in skaliranju. Na ta način smo želeli maksimizirati odzivnost komponente za samodejno skaliranje za čim hitrejše prilagajanje

spremembam in potrebam po spremembi števila primerkov posamezne mikrostoritve.

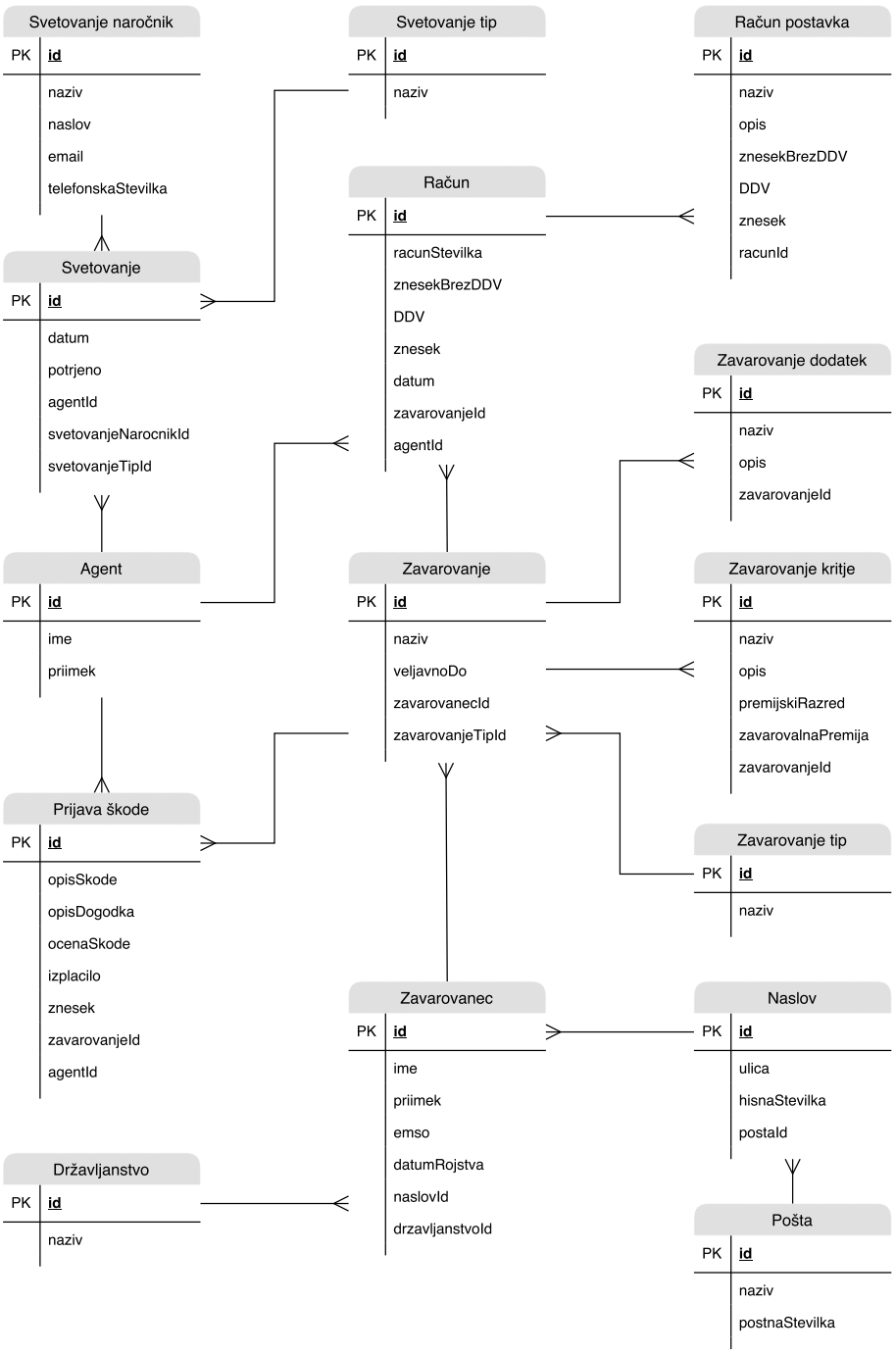
6.1.3 Zasnova in implementacija testnega primera

Za ustrezno testiranje delovanja samodejnega skaliranja z implementirano komponento smo potrebovali testni primer, ki se izvaja v vsebnikih prek orodja Kubernetes. V ta namen smo razvili aplikacijo, ki vključuje mikrostoritve, ki se med seboj različno povezujejo. Vsebujejo vmesnike REST za osnovne CRUD (*Create, Read, Update, Delete*) operacije nad pripadajočo entiteto in pridobivanje seznama entitet po vzorcu nalaganja po potrebi (ang. *lazy loading*). Mikrostoritve pri vseh klicih na zunanje mikrostoritve uporabljajo našo rešitev odpornosti na napake KumuluzEE Fault Tolerance, klice pa izvajajo prek protokola HTTP na vmesnike REST. V vseh klicih smo uporabili vzorce prekinjevalca toka, časovnika, nadomestnega mehanizma, hitrega neuspeha in pregrad. Za izpostavljanje metrik smo uporabili razširitev KumuluzEE Metrics, pri čemer smo vključili artefakt za podporo orodju Prometheus. Na storitve smo dodali našo rešitev za metrike vzorcev odpornosti na napake KumuluzEE Fault Tolerance Hystrix Metrics.

Za namen testega primera smo razvili 15 mikrostoritev, po 1 mikrostoritev za vsako entiteto. Relacijski podatkovni model lahko vidimo na sliki 6.1. Vsaka entiteta je vsebovala svojo istoimensko tabelo v svoji istoimenski podatkovni bazi. Podatkovne baze so se nahajale na sedmih strežnikih PostgreSQL[2], ki so se izvajali v ločenih strokih. Entitete smo grupirali po strežnikih PostgreSQL na naslednji način:

- Zavarovanja DB — zavarovanje.
- Zavarovanje računi DB — račun in račun postavka.
- Zavarovanci DB — zavarovanec.
- Svetovanja DB — svetovanje, svetovanje naročnik in svetovanje tip.

[2] `https://www.postgresql.org`

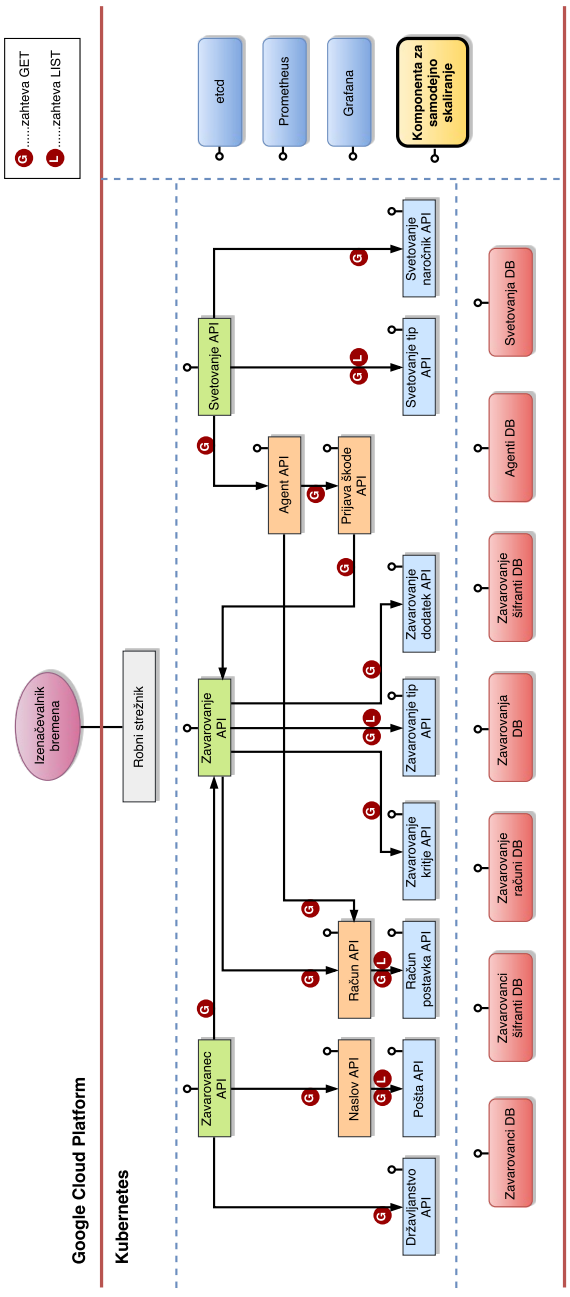


Slika 6.1: Relacijski podatkovni model testnega primera.

- Agenti DB — agent in prijava škode.
- Zavarovanje šifranti DB — zavarovanje dodatek, zavarovanje kritje in zavarovanje tip.
- Zavarovanci šifranti DB — državljanstvo, naslov in pošta.

Na sliki 6.2 lahko vidimo shemo celotnega testnega primera. V področju, označenem z oznako Kubernetes, se nahajajo vsi uporabljeni objekti postavitve, med katere spadajo mikrostoritve za pripadajoče entitete, pripadajoče podatkovne baze, podporne storitve ter orodja in komponenta za samodejno skaliranje primerkov. Samodejno smo skalirali mikrostoritve, ki pripadajo entitetam podatkovnega modela. Na shemi testnega primera lahko vidimo zasnovano povezovanje med mikrostoritvami, ki je prikazano s puščicami. Pri puščicah oznaka G predstavlja pridobivanje odvisne entitete na končni strani puščice ob prejemu zahteve tipa GET na osnovno entiteto na začetku puščice. Enako velja za zahteve tipa LIST pri oznakah L. Zahteva tipa GET na osnovni entiteti pridobiva en vnos po identifikatorju entitete, medtem ko zahteva tipa LIST predstavlja pridobivanje večjega števila vnosov osnovne entitete.

V arhitekturo testnega primera smo dodali konfiguracijski strežnik etcd za podporo konceptu odkrivanja storitev prek razširitve KumuluzEE Discovery. Strežnik etcd je hkrati služil tudi kot konfiguracijski strežnik za razširitev KumuluzEE Config. Dodali smo orodje Prometheus za zbiranje metrik in ga nastavili tako, da je povpraševalo in pridobivalo metrike iz vseh registriranih strokov. Uporabili smo orodje Grafana za grafični prikaz metrik ter dodali grafe za interaktivno nadzorovanje dogajanja v gruči. Na sliki 6.3 lahko vidimo del nadzorne plošče, prek katere smo spremljali metrike storitve samodejnega skaliranja ter metrike, pridobljene prek vzorcev odpornosti na napake. Znotraj arhitekture smo dodali zasnovano in implementirano komponento za samodejno skaliranje strokov. Implementirali smo robni strežnik, ki skrbi za preslikavo zunanjih zahtev na ustrezno mikrostoritev znotraj testne aplikacije, pri čemer se uporablja preslikava poti zahteve



Slika 6.2: Shema testnega primera za testiranje delovanja samodejnega horizontalnega skaliranja.

Slika 6.3: Videz dela nadzorne plošče orodja Grafana za nadzorovanje metrik izvajanja samodejnega horizontalnega skaliranja in metrik odpornosti na napake različnih ukazov.

na ustrezen primerek mikrostoritve. Primerek mikrostoritve se pridobiva prek odkrivanja storitev, znotraj rešitve KumuluzEE Discovery pa je poskrbljeno za izenačevanje bremena prek razvrščevalnega algoritma, imenovanega razvrščanje s krožnim dodeljevanjem (ang. *round robin*).

Za vse opisane storitve in mikrostoritve smo definirali konfiguracije objekta postavitve prek dokumenta YAML za orkestrator vsebnikov Kubernetes. V dodatku A lahko vidimo konfiguracijski dokument za mikrostoritev zavarovanje API. Gručo Kubernetes smo izvajali v javnem oblaku Google Cloud Platform prek storitve IaaS, imenovane GCE[3]. Za izvajanje gruče smo uporabili virtualno napravo tipa `n1-standard-1` (1 vCPU, 3,75 GB pomnilnika RAM) za glavno vozlišče in 7 delovnih vozlišč tipa `n1-standard-2` (2 vCPU, 7,5 GB pomnilnika RAM). Za podatkovni center smo uporabili cono `europe-west1-c`, ki se nahaja v kraju St. Ghislain v Belgiji. Uporabili smo tudi izenačevalec bremena, ki smo ga inicializirali prek tipa storitve *Load-Balancer* v orkestratorju vsebnikov Kubernetes, njegovo postavitev pa lahko

[3] `https://cloud.google.com/compute/`

vidimo v shemi testnega primera na sliki 6.2. Skrbel je za preusmerjanje zahtev na primerke robnega strežnika.

6.1.4 Pristopi k samodejnemu skaliranju in uporaba različnih ciljnih metrik

Razvito rešitev samodejnega skaliranja smo preizkušali na testnem primeru, opisanem v poglavju 6.1.3. Med razvojem smo preizkusili več metrik. Metrika števila zahtev na sekundo od metrik, ki jih zbirajo vzorci odpornosti na napake, na najbolj neposreden način sporoča informacije o obremenjenosti virov. Ko se vrednost veča, je pričakovano, da je obremenitev večja in obratno. Poleg metrike števila zahtev na sekundo smo želeli preizkusiti in poiskati način za vključevanje preostalih metrik, kot so metrike deleža napak, 50. centil in 95. centil izvajalnega časa. Pristop, ki smo ga uporabili pri ugotavljanju optimalnega števila primerkov, opisanem v poglavju 6.1.2, je bil zasnovan za metrike, kot je metrika števila zahtev na sekundo. V tem primeru tekoča povprečna vrednost zadnjih agregiranih vrednosti predstavlja ustrezno izhodišče za izračun optimalnega števila primerkov, kar pa ne velja za preostale uporabljene metrike.

Metrike deleža napak, 50. in 95. centila izvajalnega časa podajajo informacije o obremenjenosti posredno glede na trenutno število primerkov. Preveliko ali prenizko število primerkov lahko ocenimo ob odstopanjih vrednosti od pričakovanega intervala. Omenjene metrike pričakovano v zasnovani rešitvi izračunavanja števila primerkov mikrostoritve niso delovale dobro, saj nosijo drugačne vrste informacije in jih je bilo treba upoštevati drugače. Ugotovili smo, da lahko omenjene metrike uporabimo v primerih, ko pride do težav v klicani storitvi, ki je število zahtev na sekundo ne more zaznati. Povečanje deleža napak npr. sporoča določene težave v oddaljeni storitvi, na katere se lahko odzovemo s povečanjem števila primerkov in tako začasno sprostimo klicano storitev. Podobna kazalca sta tudi 50. in 95. centil izvajalnega časa, pri čemer se v tem primeru želimo odzvati na prepočasno odzivanje oddaljene storitve in izboljšati odzivne čase.

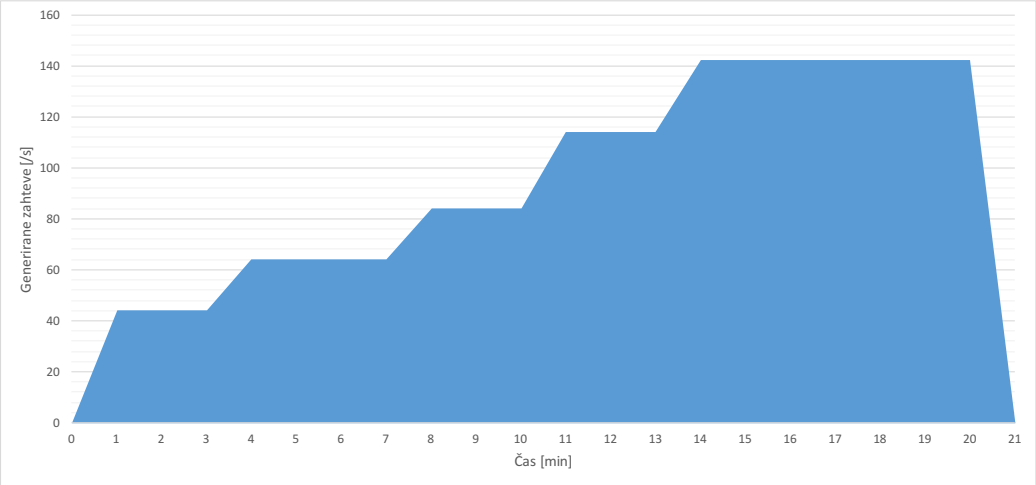
Uporaba primarne in sekundarnih kontrolnih metrik

Za upoštevanje metrik deleža napak, 50. in 95. centila izvajalnega časa smo zasnovali in uvedli dodatne sekundarne kontrolne metrike ter njihovo upoštevanje v izračunu števila primerkov mikrostoritev. To je prineslo hitrejši odziv na morebitne težave v klicani storitvi z zaznavanjem metrike deleža napak, metriki 50. in 95. centila pa sta poskrbeli za odziv na povečane odzivne čase. Metrika deleža napak se je na primer pogosto povečala ob neodzivnosti enega izmed strokov, ki ga orkestrator Kubernetes še ni zaznal kot neodzivnega. S hitro zaznavo težav in povečanjem števila primerkov smo se v takem primeru uspeli odzvati še preden so ostali kazalci pokazali težave in je orkestrator poskrbel za nadomestitev neodzivnega stroka. Na ta način smo uspeli doseči hitrejšo umiritev stanja v gruči.

Izračun števila primerkov mikrostoritve, kot je opisano v poglavju 6.1.2, smo ohranili in uporabljeno metriko preimenovali v primarno metriko — v našem primeru število zahtev na sekundo. Ob tem smo uvedli dodatni nadzor prek sekundarnih kontrolnih metrik. V poglavju 6.1.2 smo opisali metapodatkovne anotacije skaliranih objektov, med katerimi se nahajata tudi *ft-autoscale.io/secondary-keys* in *ft-autoscale.io/secondary-target-values* in definirata polje ključev metrik ter pripadajoče vrednosti za sekundarne kontrolne metrike. Sekundarne metrike upoštevamo tako, da izračunamo tekočo povprečno vrednost zadnjih n meritev za posamezno metriko. Če povprečna vrednost preseže podano vrednost za pripadajočo sekundarno metriko, komponenta samodejnega skaliranja poveča izračunano optimalno število primerkov za 1. Postopek skaliranja se nato nadaljuje s preverjanjem preseganja nastavljenih okvirjev minimalnega in maksimalnega števila primerkov ter zadnjih časov skaliranja.

6.2 Rezultati in diskusija

V tem poglavju bomo predstavili rezultate poskusov in meritev, s katerimi validiramo uspešnost naše rešitve samodejnega skaliranja prek metrik od-



Slika 6.4: Graf generiranja zahtev skozi čas izvajanja poskusa.

pornosti na napake. Opisali bomo izvedbo poskusov, zasnovo referenčnega optimalnega rezultata in primerjavo rešitve z uveljavljenim pristopom k skaliranju. Ob rezultatih bomo predstavili diskusijo o doseženih rezultatih ter možnostih nadaljnega razvoja.

Vsa testiranja samodejnega skaliranja smo izvajali s testnim primerom, opisanim v poglavju 6.1.3. Za generiranje bremena smo uporabili orodje Apache JMeter[4]. Breme smo generirali iz treh naprav, ki so generirale zahteve na vmesnike REST za pridobivanje entitet po identifikatorju in za pridobivanje seznama entitet. Vse zahteve so se pošiljale na storitev, ki je dostopna zunaj gruče Kubernetes prek izenačevalca bremena. Pri generiranju zahtev smo uporabili konstanten generator zahtev, ki je zahteve pošiljal s frekvenco 1 zahteve na sekundo, na vsaki od treh naprav pa je generator deloval v več nitih. Med opravljanjem poskusa smo postopoma povečevali število niti, ki so generirale zahteve, ter opazovali odzivanje samodejnega skaliranja in spreminjanja števila primerkov mikrostoritev. Na sliki 6.4 lahko vidimo graf generiranja zahtev in povečevanje bremena na testno aplikacijo skozi čas izvajanja poskusa.

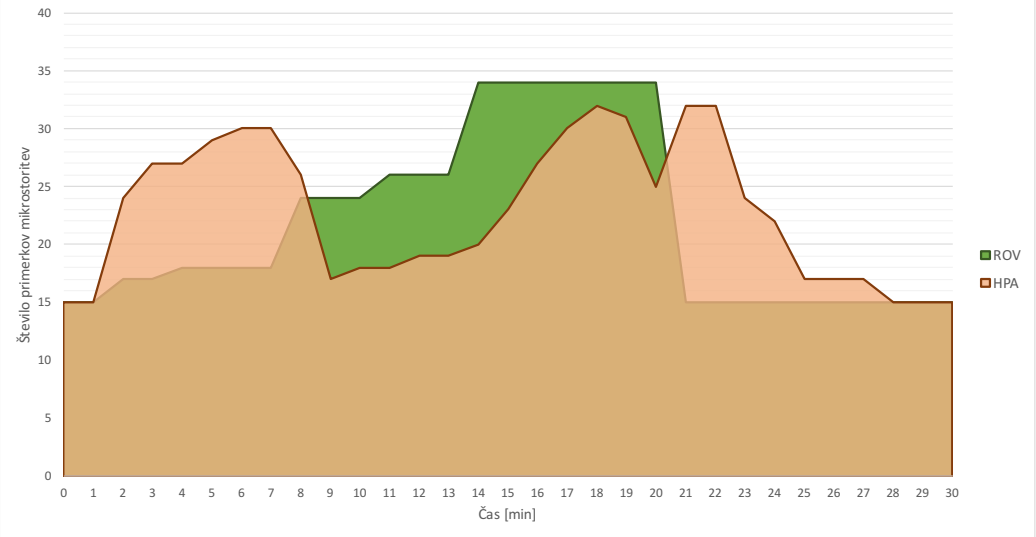
[4] `http://jmeter.apache.org`

Na podlagi generiranih zahtev na sekundo, povezovanja mikrostoritev znotraj testnega primera in končnih konfiguracij za delovanje samodejnega skaliranja smo izračunali pričakovano spreminjanje števila primerkov mikrostoritev znotraj testnega primera, ki je v poskusih predstavljal referenčne optimalne vrednosti, na osnovi katerih smo izvedli primerjavo različnih načinov samodejnega skaliranja. Končne konfiguracije so predstavljene v tabeli 6.1. Poskus za oceno delovanja komponente samodejnega skaliranja primerkov mikrostoritev prek metrik odvisnosti na napake (ONN) smo izvedli nad testnim primerom in v opisanih testnih konfiguracijah. Za primerjavo uspešnosti smo poskus izvedli tudi z uporabo objekta HPA orkestratorja Kubernetes. Objekt HPA smo v tem primeru ustvarili za vsak objekt postavitve mikrostoritev v testnem primeru. Uporabili smo omejitve procesorskih virov in pomnilnika RAM, nastavili pa smo enake vrednosti, kot so navedene v tabeli 6.1. Ciljno povprečno vrednost pri objektu HPA smo v vseh primerih omejili na 60 % obremenjenost vira CPU.

Na sliki 6.5 lahko vidimo graf primerjave samodejnega skaliranja z objektom HPA, na sliki 6.6 pa graf primerjave samodejnega horizontalnega skaliranja s komponento ONN. Oba grafa vsebujeta tudi prikaz referenčnih optimalnih vrednosti, ki služi primerjavi obeh pristopov. Ob vizualni primerjavi grafov lahko opazimo, da sta se obe metodi odzivali na povečevanje števila zahtev. Pri obeh pristopih je vidno začetno povečanje števila primerkov ob začetku generiranja zahtev. Ta je še bolj izrazit pri pristopu z objektom HPA in je posledica izrazito večje začetne porabe virov CPU. Pri pristopu s skaliranjem prek komponente ONN je začetno povečanje posledica večjih odzivnih časov zaradi začetne inicializacije povezav in notranjih aplikacijskih komponent v mikrostoritvah. Prvi odzivi v povečanju v pristopu ONN so torej posledica reagiranja na vrednosti sekundarnih kontrolnih metrik. V obeh primerih je število primerkov po začetnem skoku padlo. Opazimo lahko, da objekt HPA ni nikoli dosegel največjega števila primerkov izračunanega v referenčnih optimalnih vrednostih. Delovanje sistema zato v pristopu z objektom HPA v določenem obdobju tudi ni bilo popolnoma stabilno, saj

Tabela 6.1: Končne konfiguracije samodejnega skaliranja objektov postavitve v testnem primeru — omejitev maksimalnega števila primerkov mikrostoritve (MAX), omejitev procesorskih virov (CPU), pomnilnika (RAM), nastavitev primarne metrike števila zahtev na sekundo (RPS) in nastavitve sekundarnih kontrolnih metrik 95. centila (95C) ter deleža napak (ERR).

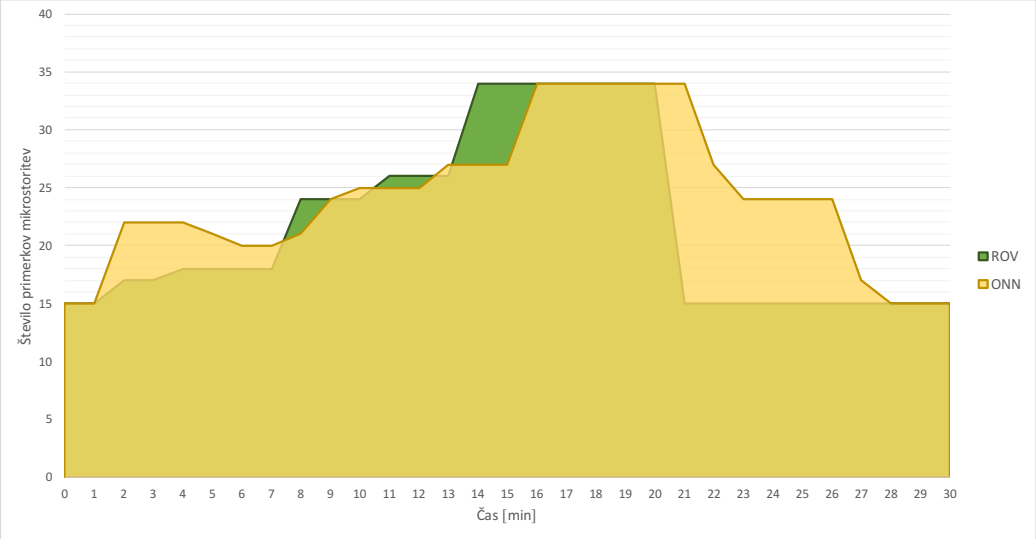
Mikrostoritev	MAX	CPU	RAM	RPS	95C	ERR
Zavarovanje	6	0.3	640 Mi	7	500 ms	10 %
Svetovanje	5	0.3	640 Mi	8	400 ms	10 %
Zavarovanec	4	0.3	640 Mi	10	400 ms	10 %
Račun	4	0.25	512 Mi	15	250 ms	10 %
Agent	3	0.25	512 Mi	15	350 ms	10 %
Prijava škode	3	0.25	512 Mi	15	300 ms	10 %
Naslov	3	0.25	512 Mi	15	200 ms	10 %
Zavarovanje tip	2	0.2	384 Mi	25	150 ms	10 %
Zavarovanje dod.	2	0.2	384 Mi	25	150 ms	10 %
Zavarovanje krit.	2	0.2	384 Mi	25	150 ms	10 %
Svetovanje tip	2	0.2	384 Mi	25	150 ms	10 %
Svetovanje nar.	2	0.2	384 Mi	25	150 ms	10 %
Državljanstvo	2	0.2	384 Mi	25	150 ms	10 %
Pošta	2	0.2	384 Mi	25	150 ms	10 %
Račun postavka	2	0.2	384 Mi	25	150 ms	10 %



Slika 6.5: Graf samodejnega skaliranja primerkov mikrostoritev z uporabo objekta samodejnega horizontalnega skaliranja strokov orkestratorja Kubernetes (HPA) v primerjavi z referenčnimi optimalnimi vrednostmi (ROV).

smo zaznali nekaj odpovedi delovanja strokov, kar je vidno v intervalu med 19. in 21. minuto v grafu na sliki 6.5. Sistem se je sicer kasneje v 21. minuti uspel stabilizirati ravno ob končanju generiranja zahtev. V obeh pristopih sicer lahko zaznamo težnjo po približevanju izračunanim referenčnim optimalnim vrednostim.

Poleg vizualne primerjave smo pristopa primerjali tudi kvantitativno prek odstopanja od referenčnih optimalnih vrednosti. Za dobljene krivulje v grafih smo izračunali površino pod krivuljo AUC (ang. *Area Under Curve*). Nato smo izračunali odstopanja v vrednostih AUC obeh pristopov od referenčnih optimalnih vrednosti kot vsoto absolutnih vrednosti razlik v površinah na posameznih odsekih. Dobljeni vrednosti smo ovrednotili kot delež odstopanja od vrednosti AUC referenčnih optimalnih vrednosti. Rezultati so prikazani v tabeli 6.2. 0-% odstopanje predstavlja referenčne optimalne vrednosti. Z rešitvijo samodejnega skaliranja primerkov mikrostoritev prek komponente ONN smo dosegli 16-% odstopanje. Objekt HPA je v enaki izvedbi poskusa



Slika 6.6: Graf samodejnega skaliranja primerkov mikrostoritev z uporabo komponente samodejnega horizontalnega skaliranja prek metrik odpornosti na napake (ONN) v primerjavi z referenčnimi optimalnimi vrednostmi (ROV).

Tabela 6.2: Rezultati primerjave pristopov z referenčnimi optimalnimi vrednostmi. Prvi stolpec predstavlja površino pod krivuljo (AUC), drugi stolpec predstavlja odstopanje kot razliko površin, tretji stolpec pa predstavlja delež odstopanja od površine referenčnih optimalnih vrednosti. Druga vrstica tabele prikazuje rezultate za referenčne optimalne vrednosti (ROV).

Pristop	AUC	Odstopanje	Delež odstopanja
ROV	659	0	0 %
HPA	693	193	29,29 %
ONN	732	106	16,08 %

dosegel 29-% odstopanje. To pomeni, da se je rešitev s komponento ONN odrezala za 13 odstotnih točk bolje od objekta HPA ter se bolj približala referenčnim optimalnim vrednostim. Rešitev samodejnega skaliranja prek metrik odpornosti na napake je v primerjavi s pristopom prek objekta HPA za 45 % izboljšala odstopanje od referenčnih optimalnih vrednosti.

Ob podrobnejšem pregledu poskusa samodejnega skaliranja s komponento ONN se je izkazalo, da so v nekaj primerih sekundarne kontrolne metrike poskrbele za hitrejšo reakcijo na povečanje bremena in s tem povečanjem števila primerkov mikrostoritve, v kateri se je zaznal povečan odzivni čas. Kasneje so v večini primerov tudi primarne metrike zahtev na sekundo potrdile ustrezno povečanje primerkov. V ostalih primerih je šlo za začetni povečan odzivni čas zaradi inicializacije aplikacij, ki so se izvajale v posameznih vsebnikih strokov in so se zvrstile ob prvih prejetih zahtevah. Začetni povečani odzivni časi so poskrbeli za prvo večje odstopanje rešitve v primerjavi z referenčnimi optimalnimi vrednostmi. Odstopanje je vidno na grafu na sliki 6.6 v časovnem intervalu med 2. in 6. minuto. Drugo večje odstopanje je bilo posledica večje časovne omejitve pred dovoljenim zmanjšanjem števila primerkov za posamezno mikrostoritev. V grafu na sliki 6.6 lahko to odstopanje opazimo v časovnem intervalu med 20. in 26. minuto. Kljub temu da je časovna

omejitev za zmanjšanje primerkov od zadnjega izvedenega skaliranja poskrbela za največji odmik od referenčne vrednosti, menimo, da tovrsten pristop ne predstavlja večje škode v samodejnem skaliranju. Dovolj hiter oziv na povečano obremenitev aplikacije je veliko bolj pomemben kot hiter odziv na zmanjšanje obremenitve. V poskusu je ta razkorak še nekoliko večji, saj smo iz maksimalnega generiranja zahtev direktno prešli v minimalno generiranje zahtev, ki je v našem primeru predstavljalo ustavitev generatorjev zahtev.

Poleg večje fleksibilnosti pri izbiri metrik, tako primarnih kot sekundarnih, je ena od prednosti zasnovane komponente ONN v primerjavi z objektom HPA tudi v tem, da rešitev ne potrebuje omejevanja virov CPU za posamezne stroke. Izkazalo se je, da je omejevanje virov CPU težavno pri razporejanju strokov na vozlišča v orodju Kubernetes. Orodje ob oblikovanju stroka preverja zasedenost virov vozlišča glede na nastavljene omejene vire strokov, ki se na vozlišču izvajajo, ne upošteva pa trenutne porabe virov vozlišča. Ustvarjanje novega stroka zato spodleti, ko stroki z omejitvijo virov na posameznem delovnem vozlišču dosežejo razpoložljive vire delovnega vozlišča. Zato smo bili primorani precej povečati število delovnih vozlišč testne aplikacije na oblačni storitvi GCE za nemoteno upravljanje z gručo. Tovrstno povečanje števila vozlišč se je posledično poznalo pri dejanski porabi virov in nenazadnje tudi na finančni plati. Z izklopom omejitve virov CPU strokov smo lahko aplikacijo izvajali na manjšem številu delovnih vozlišč, pri čemer smo lahko z zasnovano rešitvijo še vedno enako kakovostno izvajali samodejno horizontalno skaliranje strokov, medtem ko skaliranje prek objekta HPA orodja Kubernetes zaradi nenastavljenih omejitev virov ni delovalo.

Prostora za napredek v implementirani komponenti samodejnega horizontalnega skaliranja prek metrik odpornosti na napake je še veliko. Omogočiti želimo konfiguracijo več nastavitev prek konfiguracijskega sistema. Zelo dobrodošlo bi bilo spletno grafično orodje, ali v začetni fazi vsaj ukazno, prek katerega bi lahko dinamično spreminjali nastavitve delovanja in bolj kakovostno spremljali samo delovanje zasnovane rešitve samodejnega skaliranja. Smiselno bi bilo integrirati zasnovano rešitev z objektom HPA orodja Kuber-

netes. Alternativa bi bila povezava z metričnim sistemom orodja Kubernetes (Heapster), prek katerega bi lahko pri odločitvah upoštevali tudi metrike, kot so poraba virov CPU in pomnilnika RAM, ter jih združili s podatki, pridobljenimi z metrikami odpornosti na napake. Izboljšave so mogoče tudi v metodi za izračun in določitev optimalnega števila primerkov.

Poglavje 7

Zaključek

V magistrskem delu smo analizirali področje odpornosti na napake in skalabilnosti v oblačnih aplikacijah. V arhitekturo oblačnih aplikacij smo umestili prekinjevalce toka in ostale vzorce odpornosti na napake kot centralni element, s čimer smo izboljšali odpornost, prožnost, skalabilnost in fleksibilnost aplikacij. Zasnovali in implementirali smo rešitev za uporabo vzorcev odpornosti na napake za mikrostoritve v Javi. Zasnovali smo izboljšano rešitev za samodejno skaliranje na podlagi metrik odpornosti na napake, ki smo jo uspešno ovrednotili in potrdili.

Z implementacijo rešitve za uporabo vzorcev odpornosti na napake za mikrostoritve v Javi smo omogočili enostavno uporabo vzorcev odpornosti na napake. Uporabili smo dobro uveljavljeno orodje Hystrix. Orodje izvaja klice na zunanje odvisnosti prek ukazov, ki izvajajo odpornost na napake. V rešitvi smo podprli prvo končno verzijo specifikacije MicroProfile Fault Tolerance. Specifikacija določa način uporabe vzorcev in njihovo konfiguracijo. Rešitev smo zasnovali kot razširitev KumuluzEE Fault Tolerance za ogrodje za mikrostoritve KumuluzEE. Skupaj z rešitvijo za uporabo vzorcev smo omogočili dostop do metrik ukazov. Ob izvajanju ukazov se v orodju Hystrix zbira velika količina podatkov o dostopnosti in odzivnosti oddaljenih virov. Izpostavitev podatkov prek metrik smo v rešitvi omogočili kot razširitev KumuluzEE Fault Tolerance Hystrix Metrics.

Na podlagi rešitve za uporabo odpornosti na napake in pripadajočih metrik smo zasnovali izboljšano rešitev za samodejno skaliranje. V rešitvi se omenjene metrike uporabljajo kot vir za odločanje o skaliranju primerkov mikrostoritev. V okviru orkestratorja vsebnikov Kubernetes smo implementirali komponento za skaliranje prek metrik odpornosti na napake, ki skrbi za pridobivanje metrik iz primerkov mikrostoritev in ustrezno prilagajanje števila primerkov glede na trenutne potrebe. Zasnovana rešitev kot primarno metriko uporablja število zahtev na sekundo, na podlagi katere določi optimalno število primerkov oziroma strokov za posamezen objekt postavitve orkestratorja vsebnikov Kubernetes. Dodatno upošteva tudi sekundarne kontrolne metrike odpornosti na napake, kot sta metriki deleža napak ter 95. centil izvajalnega časa. V primeru preseganja podane vrednosti pripadajoče sekundarne kontrolne metrike komponenta poskrbi za dodatno povečanje števila primerkov.

Rešitev za samodejno skaliranje prek metrik odpornosti na napake smo ovrednotili in primerjali s pristopom HPA, ki predstavlja standarden način skaliranja v orkestratorju vsebnikov Kubernetes. Pristopa smo primerjali z referenčno optimalno vrednostjo. Rezultati so pokazali 16-% odstopanje naše rešitve od referenčne optimalne vrednosti, medtem ko je pristop HPA od referenčne optimalne vrednosti odstopal za 29 %. Naša rešitev se je za 13 odstotnih točk bolj približala referenčnim optimalnim vrednostim in je v zasnovanem poskusu za 45 % izboljšala skaliranje s pristopom HPA. S tem smo validirali našo rešitev, potrdili koncept skaliranja prek metrik odpornosti na napake, izboljšali samodejno skaliranje in povečali skalabilnost ter elastičnost oblačne aplikacije.

Rešitev za odpornost na napake KumuluzEE Fault Tolerance in metrike odpornosti na napake orodja Hystrix KumuluzEE Fault Tolerance Hystrix Metrics bomo še naprej razvijali. Za področje samodejnega horizontalnega skaliranja prek zasnovane komponente ostaja odprtih še veliko možnosti nadaljnjega razvoja. Mednje spadajo globlja integracija v orkestrator vsebnikov Kubernetes, povečanje možnosti konfiguracij, podpora uvoza metrik iz

sistemov za zbiranje metrik v orodju Kubernetes in integracija z objektom samodejnega horizontalnega skaliranja strokov orkestratorja vsebnikov Kubernetes.

Dodatek A

Primer konfiguracije objekta postavitve v orkestratorju vsebnikov Kubernetes prek dokumenta YAML

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: zavarovanje-api
  annotations:
    ft-autoscale.io/autoscale: 'true'
    ft-autoscale.io/replicas-min: '1'
    ft-autoscale.io/replicas-max: '6'
    ft-autoscale.io/target-value: '7'
    ft-autoscale.io/secondary-keys: 'executionTime_95,errorPercentage'
    ft-autoscale.io/secondary-target-values: '500,15'
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: zavarovanje-api
      annotations:
        prometheus.io/scrape: 'true'
        prometheus.io/probe: 'true'
        prometheus.io/port: '8080'
```

```
      prometheus.io/path: '/prometheus'
      ft-autoscale.io/scrape: 'true'
      ft-autoscale.io/port: '8080'
      ft-autoscale.io/command.zavarovanje-racun.http-get-racuni:
        'deployment/racun-api'
      ft-autoscale.io/command.zav-zav-dodatek.http-get-zav-dodatki:
        'deployment/zavarovanje-dodatek-api'
      ft-autoscale.io/command.zav-zav-kritje.http-get-zav-kritja:
        'deployment/zavarovanje-kritje-api'
      ft-autoscale.io/command.zav-zav-tip.http-get-zav-tipi:
        'deployment/zavarovanje-tip-api'
      ft-autoscale.io/command.zav-zav-tip.http-get-zav-tip:
        'deployment/zavarovanje-tip-api'
  spec:
    containers:
    - name: zavarovanje-api
      image: lukasarc/kumuluzee-fault-tolerance-zavarovanje-api
      ports:
      - containerPort: 8080
      resources:
        limits:
          memory: '640Mi'
          cpu: 0.3
      env:
      - name: KUMULUZEE_DISCOVERY_ETCD_HOSTS
        value: http://etcd:2379
      livenessProbe:
        httpGet:
          path: /v1/health
          port: 8080
        initialDelaySeconds: 60
        periodSeconds: 30
        timeoutSeconds: 1
```

Literatura

[1] A. Balalaie, A. Heydarnoori, P. Jamshidi, Migrating to cloud-native architectures using microservices: an experience report, in: European Conference on Service-Oriented and Cloud Computing, Springer, 2015, pp. 201–215.

[2] Pivotal Software, Inc., Cloud-Native — Pivotal, dostopno na: `https://pivotal.io/cloud-native` (pridobljeno: avgust 2017).

[3] M. Stine, Migrating to cloud-native application architectures (2015).

[4] C. Posta, Microservices for java developers: A hands-on introduction to frameworks and containers (2015).

[5] G. Toffetti, S. Brunner, M. Blöchlinger, J. Spillner, T. M. Bohnert, Self-managing cloud-native applications: Design, implementation, and experience, Future Generation Computer Systems (2016) – `doi:http://dx.doi.org/10.1016/j.future.2016.09.002`.
URL `http://www.sciencedirect.com/science/article/pii/S0167739X16302977`

[6] Netflix OSS, Netflix/Hystrix, dostopno na: `https://github.com/Netflix/Hystrix` (pridobljeno: avgust 2017).

[7] H. Jamjoom, D. Williams, U. Sharma, Don't call them middleboxes, call them middlepipes, in: Proceedings of the third workshop on Hot topics in software defined networking, ACM, 2014, pp. 19–24.

[8] Ben Schmaus, Making the Netflix API More Resilient, dostopno na: https://medium.com/netflix-techblog/making-the-netflix-api-more-resilient-a8ec62159c2d (pridobljeno: avgust 2017).

[9] G. Toffetti, S. Brunner, M. Blöchlinger, F. Dudouet, A. Edmonds, An architecture for self-managing microservices, in: Proceedings of the 1st International Workshop on Automated Incident Management in Cloud, ACM, 2015, pp. 19–24.

[10] F. Montesi, J. Weber, Circuit breakers, discovery, and api gateways in microservices, arXiv preprint arXiv:1609.05830.

[11] Adam Wiggins, The Twelve-Factor App, dosegljivo na: https://12factor.net (pridobljeno: avgust 2017).

[12] K. Hoffman, Beyond the twelve-factor app (2016).

[13] Lewis, James and Fowler, Martin, Microservices, dostopno na: https://martinfowler.com/articles/microservices.html (pridobljeno: avgust 2017).

[14] A. Balalaie, A. Heydarnoori, P. Jamshidi, Microservices architecture enables devops: migration to a cloud-native architecture, IEEE Software 33 (3) (2016) 42–52.

[15] D. Namiot, M. Sneps-Sneppe, On micro-services architecture, International Journal of Open Information Technologies 2 (9) (2014) 24–27.

[16] Elasticsearch, Logstash: Collect, Parse, Transform Logs — Elastic, dostopno na: https://www.elastic.co/products/logstash (pridobljeno: avgust 2017).

[17] Elasticsearch, Elasticsearch: RESTful, Distributed Search & Analytics — Elastic, dostopno na: https://www.elastic.co/products/elasticsearch (pridobljeno: avgust 2017).

[18] Elasticsearch, Kibana: Explore, Visualize, Discover Data — Elastic, dostopno na: `https://www.elastic.co/products/kibana` (pridobljeno: avgust 2017).

[19] Richardson, Chris, Application metrics, dostopno na: `http://microservices.io/patterns/observability/application-metrics.html` (pridobljeno: avgust 2017).

[20] Prometheus, Prometheus - Monitoring system & time series database, dostopno na: `https://prometheus.io` (pridobljeno: avgust 2017).

[21] Graphite, Graphite, dostopno na: `https://graphiteapp.org` (pridobljeno: avgust 2017).

[22] Richardson, Chris, Health check, dostopno na: `http://microservices.io/patterns/observability/health-check-api.html` (pridobljeno: avgust 2017).

[23] Pivotal Software, Inc., Spring Cloud, dostopno na: `http://projects.spring.io/spring-boot/` (pridobljeno: avgust 2017).

[24] C. Walls, Spring Boot in action, Manning Publications Co., 2016.

[25] Pivotal Software, Inc., Spring Cloud, dostopno na: `http://projects.spring.io/spring-cloud/` (pridobljeno: avgust 2017).

[26] Red Hat, Inc., Rightsize your Java EE Applications — WildFly Swarm, dostopno na: `http://wildfly-swarm.io` (pridobljeno: avgust 2017).

[27] Oracle, What Is an Enterprise Bean? - The Java EE 6 Tutorial, dostopno na: `http://docs.oracle.com/javaee/6/tutorial/doc/gipmb.html` (pridobljeno: september 2017).

[28] The Eclipse Foundation, microprofile.io, dostopno na: `http://microprofile.io` (pridobljeno: avgust 2017).

[29] Sutter, Kevin and Blevins, David, Eclipse MicroProfile 1.2, dostopno na: https://drive.google.com/file/d/0B7PK4ZIk1mjAUTJBem1FWTEwdDQ/view (pridobljeno: september 2017).

[30] The Eclipse Foundation, microprofile.io - Configuration for MicroProfile, dostopno na: http://microprofile.io/project/eclipse/microprofile-config (pridobljeno: avgust 2017).

[31] S. Newman, Building microservices: designing fine-grained systems, "O'Reilly Media, Inc.", 2015.

[32] M. Richards, Microservices antipatterns and pitfalls (2015).

[33] M. Eisele, Developing reactive microservices: enterprise implementation in java.

[34] Immobiliare Labs, Apache Kafka producers are not fault tolerant, dostopno na: https://labs.immobiliare.it/kafka-producers-are-not-fault-tolerant/ (pridobljeno: avgust 2017).

[35] M. Nygard, Release it!: design and deploy production-ready software, Pragmatic Bookshelf, 2007.

[36] Fowler, Martin, CircuitBreaker, dostopno na: https://martinfowler.com/bliki/CircuitBreaker.html (pridobljeno: avgust 2017).

[37] Netflix OSS, How To Use · Netflix/Hystrix Wiki, dostopno na: https://github.com/Netflix/Hystrix/wiki/How-To-Use (pridobljeno: avgust 2017).

[38] A. Robertson, Resource fencing using stonith, White Paper, August.

[39] Netflix OSS, How it Works · Netflix/Hystrix Wiki, dostopno na: https://github.com/Netflix/Hystrix/wiki/How-it-Works (pridobljeno: avgust 2017).

[40] jhalterman, jhalterman/failsafe: Simple, sophisticated failure handling, dostopno na: `https://github.com/jhalterman/failsafe` (pridobljeno: avgust 2017).

[41] resilience4j, resilience4j/resilience4j: Resilience4j is a fault tolerance library designed for Java8 and functional programming, dostopno na: `https://github.com/resilience4j/resilience4j` (pridobljeno: avgust 2017).

[42] resilience4j, Resilience4j User Guide, dostopno na: `http://resilience4j.github.io/resilience4j/#_bulkhead` (pridobljeno: avgust 2017).

[43] The Eclipse Foundation, microprofile.io, dostopno na: `http://microprofile.io/project/eclipse/microprofile-fault-tolerance` (pridobljeno: september 2017).

[44] Oracle Corporation, InterceptorBinding (Java EE 6), dostopno na: `http://docs.oracle.com/javaee/6/api/javax/interceptor/InterceptorBinding.html` (pridobljeno: avgust 2017).

[45] Kumuluz, Configuration · kumuluz/kumuluzee Wiki, dostopno na: `https://github.com/kumuluz/kumuluzee/wiki/Configuration` (pridobljeno: avgust 2017).

[46] M. Šadl, Statistika : gradivo za 1. letnik, Zavod IRC, 2008.

[47] Coda Hale, Yammer Inc., Metrics Core — Metrics, dostopno na: `http://metrics.dropwizard.io/3.2.3/manual/core.html` (pridobljeno: september 2017).

[48] The Apache Software Foundation, Maven – Introduction to the Dependency Mechanism, dostopno na: `https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html` (pridobljeno: september 2017).

[49] A. B. Bondi, Characteristics of scalability and their impact on performance, in: Proceedings of the 2nd international workshop on Software and performance, ACM, 2000, pp. 195–203.

[50] L. M. Vaquero, L. Rodero-Merino, R. Buyya, Dynamically scaling applications in the cloud, ACM SIGCOMM Computer Communication Review 41 (1) (2011) 45–52.

[51] H. Kang, M. Le, S. Tao, Container and microservice driven design for cloud infrastructure devops, in: Cloud Engineering (IC2E), 2016 IEEE International Conference on, IEEE, 2016, pp. 202–211.

[52] A. M. Joy, Performance comparison between linux containers and virtual machines, in: Computer Engineering and Applications (ICACEA), 2015 International Conference on Advances in, IEEE, 2015, pp. 342–346.

[53] D. Bernstein, Containers and cloud: From lxc to docker to kubernetes, IEEE Cloud Computing 1 (3) (2014) 81–84.

[54] Canonical Ltd., Linux Containers, dostopno na: `https://linuxcontainers.org` (pridobljeno: avgust 2017).

[55] J. Baier, Getting Started with Kubernetes, Packt Publishing Ltd, 2015.

[56] jancorg, LibContainer Overview - I left my leg in Jaglan Beta, dostopno na: `http://jancorg.github.io/blog/2015/01/03/libcontainer-overview/` (pridobljeno: september 2017).

[57] C. Richardson, F. Smiht, Microservices from design to deployment (2016).

[58] The Linux Foundation, Kubernetes - Production-Grade Container Orchestration, dostopno na: `https://kubernetes.io` (pridobljeno: avgust 2017).

[59] Docker Inc., Swarm Mode Overview — Docker Documentation, dostopno na: `https://docs.docker.com/engine/swarm/` (pridobljeno: avgust 2017).

[60] Docker Inc., Overview of Docker Compose — Docker Documentation, dostopno na: `https://docs.docker.com/compose/overview/` (pridobljeno: avgust 2017).

[61] The Linux Foundation, Kubernetes documentation - Kubernetes, dostopno na: `https://kubernetes.io/docs/home/` (pridobljeno: avgust 2017).

[62] B. Burns, K. Hightower, J. Beda, Kubernetes: Up and Running, "O'Reilly Media, Inc.", 2017.

[63] The Linux Foundation, Services - Kubernetes, dostopno na: `https://kubernetes.io/docs/concepts/services-networking/service/` (pridobljeno: avgust 2017).

[64] Autoscaling - Kubernetes, Autoscaling - Kubernetes, dostopno na: `https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/` (pridobljeno: avgust 2017).

[65] Kubernetes, community/horizontal-pod-autoscaler.md at master · kubernetes/community, dostopno na: `https://github.com/kubernetes/community/blob/master/contributors/design-proposals/horizontal-pod-autoscaler.md` (pridobljeno: avgust 2017).